Conditioning Graphs: Practical Structures for Inference in Bayesian Networks

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Kevin John Grant

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Probability is a useful tool for reasoning when faced with uncertainty. Bayesian networks offer a compact representation of a probabilistic problem, exploiting independence amongst variables that allows a factorization of the joint probability into much smaller local probability distributions.

The standard approach to probabilistic inference in Bayesian networks is to compile the graph into a join-tree, and perform computation over this secondary structure. While join-trees are among the most time-efficient methods of inference in Bayesian networks, they are not always appropriate for certain applications. The memory requirements of join-tree can be prohibitively large. The algorithms for computing over join-trees are large and involved, making them difficult to port to other systems or be understood by general programmers without Bayesian network expertise.

This thesis proposes a different method for probabilistic inference in Bayesian networks. We present a data structure called a *conditioning graph*, which is a runtime representation of Bayesian network inference. The structure mitigates many of the problems of join-tree inference. For example, conditioning graphs require much less space to store and compute over. The algorithm for calculating probabilities from a conditioning graph is small and basic, making it portable to virtually any architecture. And the details of Bayesian network inference are compiled away during the construction of the conditioning graph, leaving an intuitive structure that is easy to understand and implement without any Bayesian network expertise.

In addition to the conditioning graph architecture, we present several improvements to the model, that maintain its small and simplistic style while reducing the runtime required for computing over it. We present two heuristics for choosing variable orderings that result in shallower elimination trees, reducing the overall complexity of computing over conditioning graphs. We also demonstrate several compile

and runtime extensions to the algorithm, that can produce substantial speedup to the algorithm while adding a small space constant to the implementation. We also show how to cache intermediate values in conditioning graphs during probabilistic computation, that allows conditioning graphs to perform at the same speed as standard methods by avoiding duplicate computation, at the price of more memory. The methods presented also conform to the basic style of the original algorithm. We demonstrate a novel technique for reducing the amount of required memory for caching.

We demonstrate empirically the compactness, portability, and ease of use of conditioning graphs. We also show that the optimizations of conditioning graphs allow competitive behaviour with standard methods in many circumstances, while still preserving its small and simple style. Finally, we show that the memory required under caching can be quite modest, meaning that conditioning graphs can be competitive with standard methods in terms of time, using a fraction of the memory.

Acknowledgements

A graduate degree is not an individual effort. It is the collaboration of many individuals, direct and indirect. I mention a few here, but am grateful to all who were a part of this.

First, I wish to thank my supervisor, Michael Horsch. Mike was that supervisor that every grad student hopes for. He always had time and patience to discuss ideas, both good and bad. His knowledge of my research topics helped to keep me progressing. Most of all, I am thankful for his friendship. His faith in me was always an encouragement. Thanks Mike.

My graduate studies program was funded in large part by the National Sciences and Engineering Research Council of Canada (NSERC). Their support is greatly appreciated.

I am grateful for the guidance of my thesis committee members, whom include Eric Neufeld, Mark Keil, Winfried Grassmann, Mik Bickis, and Eugene Santos. Thank you for your help.

I would like to express my gratitude to the people of our department. I would like to thank our department heads (Jim Greer and Kevin Schneider) for giving me the opportunity to teach. During my study, I have called upon the expertise of many members of our faculty; thank you all for your help. I would also like to acknowledge the quiet, tireless efforts of our support staff and office staff, for your help and patience in making sure my computers ran, my papers printed, and my deadlines were met. Finally, a special thanks to Eric Neufeld. When Mike left on sabbatical, I often called upon Eric for his expertise in academic matters, and he always made time to answer my myriad of questions.

I could not have done this without my family. To Kelly, Jessie, Stephanie and Aaron, my grandparents, and Maureen's family (to name only a few), thanks for all your support. I especially wish to thank my parents. Their unfailing love, support, friendship, and encouragement is the reason for any successes that I may enjoy.

Most of all, I wish to thank Maureen and Samantha. Maureen has given so much during my study, and taken so little in return. Postgraduate education entails sacrifices, and the two of you have made them without question.

Saskatoon, Saskatchewan — Kevin John Grant

January 9, 2007

To my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In real world applications, rational agents, whether natural or artificial, rarely have access to full information about their environment. This information may be fundamental to making decisions and choosing actions, the role of any agent. Many physical and temporal obstructions exist, making information unobservable. One way to deal with uncertainty is to employ probability over events that are not directly observable. Probability is a popular measure of belief in instances of uncertainty.

Bayesian networks [33, 50] are a knowledge representation tool used to represent information about a problem in which there is uncertainty. Computing the probability of an event in a Bayesian network is a task often referred to as *inference*. Although computing exact probabilities from a Bayesian network is NP-hard [12], many algorithms have been designed to exploit certain properties of these networks, allowing efficient calculation of probabilities in many cases [20,35,40,50,57,65]. Most inference algorithms exploit the independencies of a probabilistic model to compute probabilities efficiently. Over the last two decades, Bayesian networks have proven themselves successful in applications requiring decision making under uncertainty, including medical diagnosis [7], classification [29], forecasting [2], and fault diagnosis [41] (Neapolitan [46] gives an excellent survey on current Bayesian network applications.)

Regardless of how probabilities are computed, however, the abstraction of a problem involving probability can be quite simple: given a context of the universe, compute the probability of a particular event. Abstraction is a popular concept in computer science. Programmers can use libraries without ever considering their details. This approach increases programming efficiency; it relieves the programmer of hav-

ing to code the component, and the library implementation is typically a good one. Abstraction allows a programmer to use libraries armed only with the knowledge of *what* it does, without knowing *how* it does it.

This abstraction occurs in Bayesian network software as well. Several commercial and academic software packages have been written that absolve programmers of most details of inference. The programmer need only supply a Bayesian network representing the problem, and the inference engine will calculate probabilities over the events of interest.[1] While these software packages provide a good solution for many users, they are not universally applicable. Some reasons that may preclude their use include the following:

1. Most software packages provide extensive services with their product for performing other kinds of inference (e.g., most probable explanation [50], sensitivity analysis [10], etc). While these extra services are useful for some applications, they tend to bloat the software, which gives the program and its available libraries a non-trivial memory footprint.

2. Most Bayesian network software libraries compile the network to a secondary structure called a junction-tree (junction-trees will be discussed in Chapter 2). While computing over junction-trees is a time-efficient means of calculation, its space requirements can be prohibitive for some problems.

3. Most software packages and their libraries are written for standard operating systems and programming languages, excluding more primitive environments such as embedded systems. Porting these applications to another environment would be time-consuming and error-prone.

4. The complexity of the software makes it difficult to get time and space guarantees. While most software can make rough predictions on time and space, they typically do not provide it in terms of number of low-level operations,

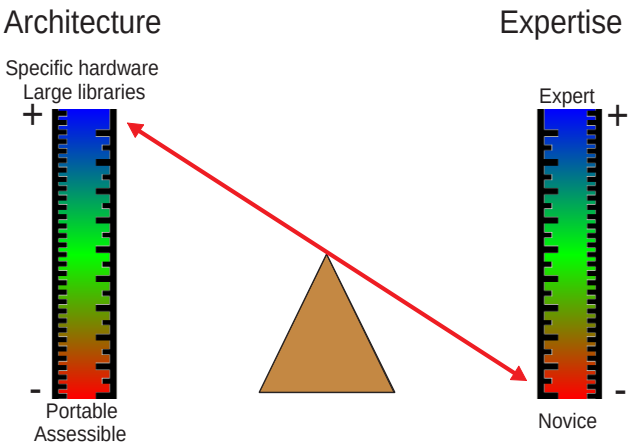
[1] Throughout the rest of this document, we assume the existence of the Bayesian network. The acquisition of a Bayesian network can be done in a number of ways (human expert, learning algorithms), but is unimportant to the concepts of this paper.

or number of bytes. This could be critical for real-time or memory-limited applications.

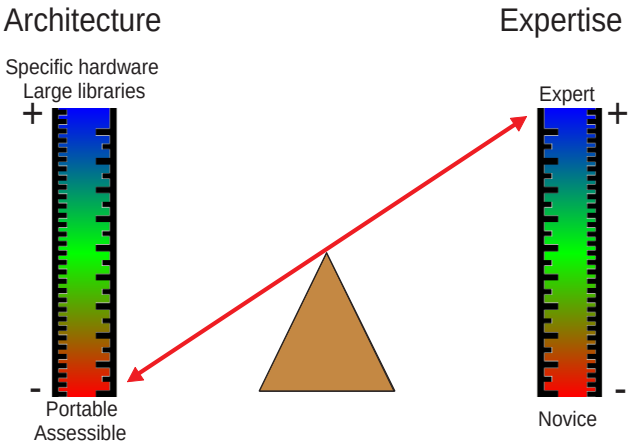
When an existing software package cannot be used, a user has no choice but to implement their own inference engine (and network representation). At this point, abstraction must be foresaken — the user must consider the details in order to implement; such an implementation assumes at least basic expertise in the area. Probability itself is a universal tool, and should not be considered a concept exclusive to experts in probability calculus. For instance, an 80% prediction of bad weather is typically enough to alter anyone's picnic plans, regardless of mathematical background. A simple fault diagnosis algorithm may only report a problem if the probability of a fault exceeds a threshold. A navigational unit may choose its direction based on a probabilistic assessment of its location. A virtual opponent in a computer game may choose its actions against the user based on the statistics of previous game-play. So while the use of probability is general, calculating probabilities from a Bayesian network still seems the domain of experts, or those with sufficient architecture and software libraries (and some basic knowledge of Bayesian network principles).

The aforementioned problems suggest a tradeoff between flexibility and expertise in Bayesian networks. Figure 1.1 demonstrates this visually. When a programmer lacks knowledge of Bayesian network inference, then third-party software must be used, with its hardware constraints and large libraries mentioned previously (Figure 1.1(a)). On the other hand, when a programmer wishes to implement a Bayesian network application on a system not supported by existing software, that programmer will have to program the inference engine, which requires expertise in that domain (Figure 1.1(b)). Expertise is also required in situations where the user needs precise knowledge of the amount of time/memory the application is going to take (a concept referred to in this document as *assessibility*).

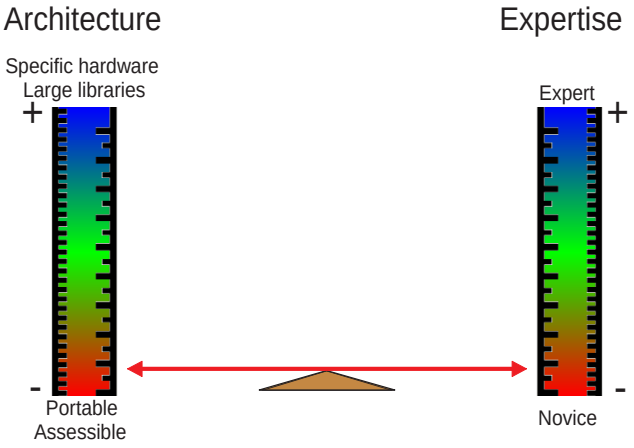
In this thesis, we consider a different type of abstraction than is typical of Bayesian network software. Abstraction typically ignores the implementation details of code. This is important, as it allows many lines of code to be summarized



(a) Novice users must use third-party software, which may have hardware constraints and large libraries.



(b) Programmers implementing inference algorithms require some background in Bayesian network technology.



(c) With conditioning graphs, the tradeoff is reduced.

Figure 1.1: The tradeoff between flexibility and expertise in Bayesian network software.

through a small, easy to understand interface. As mentioned, such an abstraction works fine if no implementation or understanding of the code is required. However, while the concepts of Bayesian network inference are reserved to those with expertise, programming constructs (conditional execution, control flow) are a universal language amongst all programmers. If we can compile inference into a series of simple programming constructs, then it should be accessible to any programmer, regardless of background. Hence, rather than presenting the user with concepts in an abstract fashion, we present them in terms of code. Such a system would allow programmers without Bayesian network expertise to implement software to compute over Bayesian networks, effectively reducing the tradeoff discussed previously (Figure 1.1(c)).

1.1 Overview

As a first step in overcoming some of the aforementioned barriers to using a Bayesian network, we compile it into a secondary structure called a *conditioning graph*. The graph is based on a tree structure called an *elimination tree*, extending by adding secondary arcs from internal nodes to leaves. The inference algorithm computes over this graph, rather than the original network, in a simple depth-first traversal that should be very accessible to programmers.

Conditioning graphs abstract away the need for expertise in Bayesian networks. Both the conditioning graph and its inference algorithm are presented using low-level programming constructs. The graph itself is represented by primitive data: integers, pointers, and floating point numbers. While these elements correspond to the local distributions and context of the Bayesian network, such details are abstracted away. The inference algorithm is so small that its memory footprint is negligible, and it is simple enough to be implemented on any architecture (no special libraries or abstract data types required).

The conversion of the Bayesian network to a conditioning graph is considered a compilation step, or offline computation; such a compilation step when performing

inference over Bayesian networks is common. Offline computation is not considered part of the final program, and therefore its resource requirements can be ignored. For junction-tree algorithms [40], the conversion of a Bayesian network to a junction-tree is considered a compile-time step. For Variable Elimination [23, 65], finding a good elimination ordering is considered compile-time and not part of the inference problem.[2] For recursive conditioning [15], construction of the dtree is compile-time.

Conditioning graphs are designed to address the problems described in the previous section. These problems, along with our proposed solution, are categorized according to the requirement that they address:

1. *Space Complexity.* The space complexity of inference in Bayesian networks using the most common inference techniques (junction-tree and variable elimination) is exponential on the treewidth of the network. Even moderately-sized networks can test the limits of standard computers, and some larger Bayesian networks are too large to compute even on high-performance machines. In contrast, our model uses conditioning, and the extra structure required for the model, in addition to CPT storage, is linear in the number of variables in the network.

2. *Portability.* Current software libraries are designed for common operating systems and programming languages. Support for other architectures is limited. The conditioning graph structure is compact and consists entirely of primitive data types. The inference algorithm is written with low-level programming instructions common to most languages. These details make conditioning graphs portable to any architecture.

3. *Memory Footprint.* Commercial software libraries for inference are typically large. However, the amount of memory required to perform inference over a Bayesian network is usually much more than the space required to store the

[2]This is considered compile-time if the same elimination order is used for all queries. Some authors advocate a dynamic elimination ordering based on the current context of the problem [55], in which case, computing an elimination order is no longer a compile-time problem.

programs; hence, the space contributions of these programs are usually ignored. However, such implementations can become a factor, especially in applications where resources are limited. Our inference algorithm requires only several lines of code; the storage of our algorithm requires trivial amounts of space.

4. *Accessibility.* Inference in Bayesian networks involves specific terminology and operations, such as marginalization, normalization, observation, all of which require some background for understanding. This is further complicated by the fact that most software packages compile the Bayesian network to a junction-tree, which requires even further expertise (triangulation, message-passing, etc). Conditioning graphs eliminate the Bayesian network-specific details, leaving the user with an easily accessible structure written in generic programming constructs.

5. *Assessibility.* The simple design of the conditioning graph, along with the succinct nature of its inference algorithm, allow for an accurate prediction of *exactly* how much memory will be required, in terms of *bytes*. Knowing exactly how much memory is required is advantageous when space is limited. Also, the simplicity of the model makes it easy to interpret this information, even for a non-expert.

   These same arguments apply to time. The number of floating point operations or recursive calls can be easily and quickly quantified by a general user. And because the algorithm is compact, its compilation is sufficiently small so as to allow for its operations to be easily countable, making it possible for a very accurate time assessment.

6. *Abstraction.* As mentioned, abstraction relieves a user of any details that are unnecessary to use a library. By making the details of the computation accessible to any user, we in effect remove some of this abstraction. However, in some cases, the abstraction of detail is still necessary. While our implementation is accessible by almost any programmer, the conditioning graph structure is intended to be automatically compiled from a Bayesian network, and an interface

is provided to maintain software engineering principles of abstraction.

7. *Time.* The complexity of inference in Bayesian networks is NP-hard [12, 14]. In addition, while conditioning algorithms require less space than junction-tree and variable elimination, they are typically less time-efficient when compared to the latter. While we cannot avoid exponential runtimes in all cases, we show how our model can exploit well-known application-specific independence, such as d-separation [50] and barren variables [59]. These optimizations can make our model competitive with standard algorithms in many circumstances. As well, because our model is a recursive decomposition, we can use caching [15] that can offer a speedup given extra amounts of space. In other words, while the space required to store conditioning graphs is much less than is used by standard inference algorithms, it does not have to be, and we can take advantage of extra space to reduce runtimes.

The techniques that we discuss in this thesis will allow a user of Bayesian networks to compile a sophisticated probabilistic model into a compact and simple component; compact enough so that the model and the inference algorithm can be implemented in a memory-restricted environment (e.g., cameras, cell phones, appliances, etc.), and simple enough to be accessible by most programmers. Bayesian networks can no longer be considered impractical to use.

1.2 Contributions and Outline

The outline of the remainder of this document is as follows. Chapter 2 describes background work upon which conditioning graphs are built. This includes a review of inference in Bayesian networks. We focus on conditioning algorithms, specifically recursive decomposition algorithms [11, 15], as the conditioning graph structure is a variant of other recursive decompositions. We also review junction-tree and variable elimination, for comparison purposes. We also show low-level precompiled inference structures (Query-DAGs [18] and Arithmetic Circuits [16]), as they represent a similar method for abstrating away the details of inference.

The primary contribution of this disseration is the conditioning graph architecture. The underlying structure of a conditioning graph, its elimination tree, is a variation of a dtree [15, 44]; its specific differences are outlined in Chapter 3 and 4. The function for calculating probabilities from the structure is a variation of the Recursive Conditioning algorithm [15], modified to compute over elimination trees. However, ours is a low-level approach, presented in such a way as to abstract away any Bayesian network-specific details, and disambiguate the programming of the structure in general.

Chapter 4 presents methods for balancing elimination trees, in order to optimize the runtime of conditioning graphs. We present a conversion process from dtrees [15] to elimination trees, in order to take advantage of dtree balancing methods. We also present a new set of heuristics for finding good elimination orderings, and empirically show that these heuristics produce more efficient elimination trees than previous approaches. We also contribute two other optimizations for the conditioning graph architecture in Chapter 4.

Chapter 5 presents compile-time optimizations to the conditioning graph structure that exploit knowledge of evidence variables and query variables. We show how sensor variables (variables that will always have an observed value) can be separated from the graph, such that we can reduce computation at runtime. We also show how to perform partial elimination in conditioning graphs to remove certain variables from the computation (i.e., variables that will not be observed or queried), also improving runtime performance. While other elimination/conditioning hybrids have been proposed, they have not yet been demonstrated in elimination trees.

Chapter 5 also presents runtime optimizations to the conditioning graph. We demonstrate a novel technique for maintaining evidence variable separation from the elimination tree dynamically. As well, we show how to exploit the well-known independence of d-separation [50] and barren variables [59] in conditioning graphs, incurring only linear time and space cost.

Chapter 6 considers caching in conditioning graphs. Caching in dtrees [11, 15] allows computation of probabilities to be as fast as elimination algorithms, at the

price of exponential memory. However, partial caching methods [4] and cache pruning methods [3] can reduce the memory costs of caching, while still providing speedup. In this chapter, we will demonstrate how these methods can be applied to conditioning graphs, both as a separate optimization, and in combination with those from Chapter 5. We also present a new approach for pruning the domains of the caches that considerably reduces the amount of memory required. The resulting model allows the calculation of probabilities with the same time complexity as the current standard algorithms, but with a fraction of the space.

The contributions of this thesis, and the future directions of this research, are summarized in Chapter 7.

Chapter 2

Background and Related Work

This chapter presents the necessary background knowledge to understand the methods and motivations of this disseration. It begins with a light review of Bayesian networks, including their structure, properties, and semantics. This beginning section also introduces the terms and notations that will be used for the remainder of the document.

The remaining sections of the chapter are devoted to reviewing inference techniques for Bayesian networks. The first section reviews the more popular methods of inference: junction-tree propagation (JTP) and variable elimination (VE). The methods of this thesis do not rely directly on these methods. However, we include them both for completeness and comparison, which we feel is important given their prevalence at the time of writing.

The second section reviews inference methods employing conditioning. Conditioning is a "reasoning by cases" approach [15], and its conservative use of memory makes it an attractive option for large, highly connected networks, situations where JTP and VE use large amounts of space. We focus primarly on recursive decompositions, as our conditioning graph is a recursive data structure, and its close relationship allows us to borrow from these previous techniques in their design.

The final section reviews some previous methods for Bayesian network compilation methods that compile away the details of inference offline. The first, Query-DAGs (Q-DAGs) [18] represents an inference operation as an arithmetic equation, parameterized by the evidence (in graphical form). The second, Arithmetic Circuits (ACs) [16] are a similar compilation that allow computations of derivatives from which values of interest (posterior probabilities, sensitivity) can be derived in

constant time.

2.1 Probability and Bayesian Networks

We denote random variables with capital letters (eg. X, Y, Z), and sets of variables with boldfaced capital letters $\mathbf{X} = \{X_1, ..., X_n\}$. Each random variable V has an associated domain $\mathcal{D}(V) = \{v_1, ..., v_k\}$. Only finite discrete variable domains are considered in this document. An *instantiation* of X, which is the assignment of variable X to a value x in its domain, is denoted $X = x$, or x for short. A *context* over a set of variables $\mathbf{X} = \{X_1, ..., X_k\}$ is the conjunction of an instantiation of each variable in $\mathbf{X}$, and is denoted $\mathbf{X} = \mathbf{x}$ or $\mathbf{x}$ for short. The set of all possible contexts over a set $\mathbf{X}$ is denoted as $\mathcal{D}(\mathbf{X})$. The size of a set $\mathbf{X}$ is denoted by $|\mathbf{X}|$.

We will denote a distribution over a set of variables using function notation (e.g. $f(\mathbf{X})$). In cases where it is clear that the notation refers to a function, we may omit the parentheses (e.g. f). We will overload the term *domain* to refer to the set of variables over which a function is defined (in addition to referring to the values of a random variable).

A *Bayesian network* [50] is a tuple $\langle \mathcal{G}, P \rangle$, where $\mathcal{G} = \langle \mathbf{X}, \mathbf{A} \rangle$ is a directed acyclic graph (DAG), $\mathbf{X} = \{X_1, ..., X_n\}$ is a set of random variables, $\mathbf{A}$ (arcs) represents direct causal influences between the variables, and P is a probability distribution over $\mathbf{X}$, such that for each $X_i \in \mathbf{X}$, instantiating its parent set in the DAG (denoted Π_i) renders X_i probabilistically independent of its graph nondescendents (denoted ND_i):

$$\forall \mathbf{Y} \subseteq ND_i \quad P(X_i|\mathbf{Y}, \Pi_i) = P(X_i|\Pi_i) \tag{2.1}$$

Since $\mathcal{G}$ has no directed cycles, it imposes a *partial ordering* on the variables of $\mathbf{X}$ [64]. Formally, if X_i is an ancestor of X_j in the DAG, then X_i must come before X_j in any ordering consistent with the partial ordering. Assume without loss of generality that the node ordering $X_1, ..., X_n$ is a total ordering consistent with the partial ordering of $\mathcal{G}$. By the definition of conditional probability, the joint probability P can be

rewritten terms of conditional probabilities:

$$P(X_1, ..., X_n) = P(X_n|X_{n-1}, ..., X_1)P(X_{n-1}, ..., X_1) \tag{2.2}$$

which can be recursively factorized into:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|X_{i-1}, ..., X_1). \tag{2.3}$$

Equation 2.1 can be substituted into Equation 2.3 to obtain the following:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|\Pi_i). \tag{2.4}$$

Stated another way, the joint probability distribution can be represented as a product of local probability distributions, called conditional probability tables (CPTs). The space complexity of this factorized representation is exponential only on the largest family (variable plus parent set), which in the worst case has the same space complexity as the entire joint probability table, but is typically much smaller.

An example of a Bayesian network is given in Figure 2.1. Its purpose is to represent the following fictitious knowledge [40]:

> Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer or bronchitis, or none of them, or more than one of them. A recent visit to Asia increase the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and bronchitis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis, as neither does the presence or absence of dyspnoea.

Each of the 8 variables in the graph are binary. To represent the joint probability $P(V, S, T, L, B, C, X, D)$ would require a table of $2^8 = 256$ values. The factorization of the joint according to the network is

$$P(V)P(S)P(T|V)P(L|S)P(B|S)P(C|T, L)P(X|C)P(D|C, B)$$

which requires only 36 values, or 14% of the original.

Given a Bayesian network, a common goal is to compute the *posterior probability*

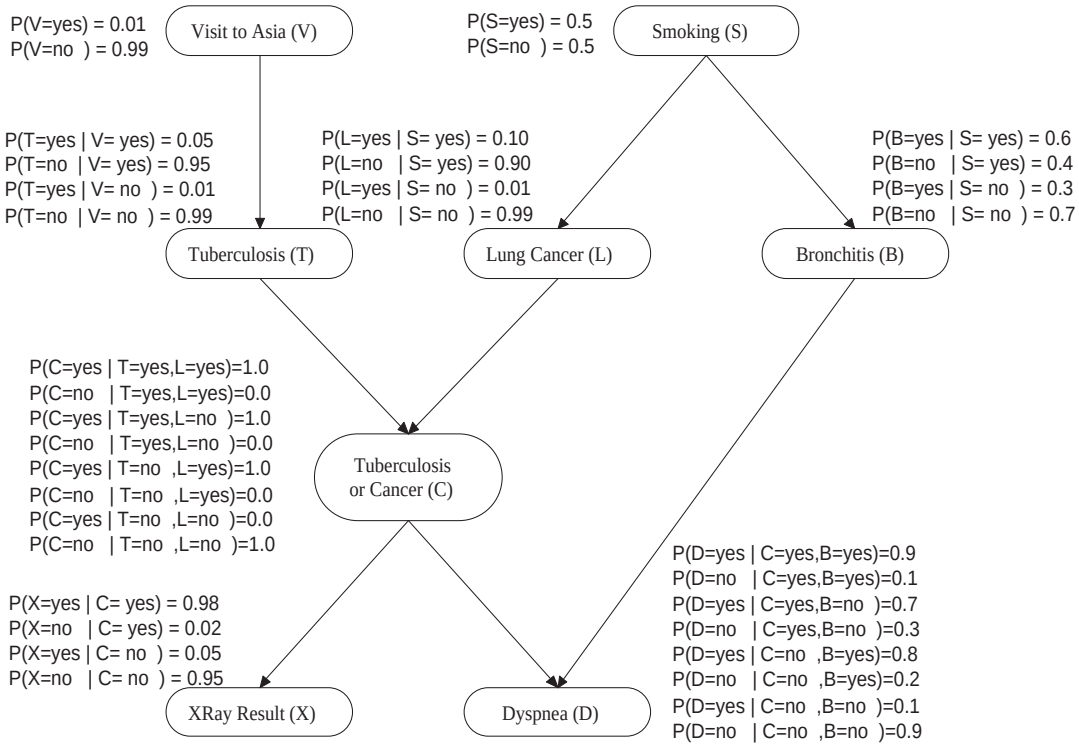


Figure 2.1: The *Asia* network, an example of a small Bayesian network [40].

distribution, or *belief*, of a variable given a context over some variables in the network (called *evidence*). For example, given the *Asia* network in Figure 2.1, one might be interested in the probability that a patient has lung cancer in light of a positive X-ray result. Calculating a posterior probability distribution over a variable or set of variables in a Bayesian network is a task often referred to as *inference*.

Formally, let $\mathbf{X}$ be a set of variables from a Bayesian network. Let $\mathbf{E} = \mathbf{e}$ be a context over a subset of $\mathbf{X}$. Finally, let X_i be a variable in $\mathbf{X}$ that is not in $\mathbf{E}$. The posterior probability distribution over X_i, given that $\mathbf{E} = \mathbf{e}$, is defined as follows:

$$P(X_i|\mathbf{E} = \mathbf{e}) = \alpha \sum_{\mathbf{x}' \in \mathcal{D}(\mathbf{X}')} P(\mathbf{x}', X_i, \mathbf{e}) \tag{2.5}$$

where $\mathbf{X}' = \mathbf{X} - (\mathbf{E} \bigcup \{X_i\})$, and α is the normalizing constant $P(\mathbf{E} = \mathbf{e})^{-1}$. For readability, when summing out a variable X_i from a distribution, we will often use the notation X_i in place of $X_i = x_i$; this allows us to write the above equation as:

$$P(X_i|\mathbf{E} = \mathbf{e}) = \alpha \sum_{\mathbf{X}'} P(\mathbf{X}', X_i, \mathbf{e}) \tag{2.6}$$

When calculating the posterior probability distribution over a variable X_i (or set of variables $\mathbf{X}$), we will refer to X_i ($\mathbf{X}$) as the *query variable(s)*, or simply query for short.

The factorization of the joint distribution according to the Bayesian network reduces the complexity of inference. Given distributions f and g, denote by $dom(f)$ the variables over which f is defined. If $X_j \notin dom(f)$ then the following equality holds [37]:

$$\sum_{X_j} f \cdot g = f \cdot \sum_{X_j} g \tag{2.7}$$

Consider the substitution of Equation 2.4 into Equation 2.6:

$$P(X_i|\mathbf{e}) = \alpha \sum_{\mathbf{X}'} \prod_{i=1}^{n} P(X_i|\Pi_i, \mathbf{e}). \tag{2.8}$$

Let $\mathbf{Y_j}$ be a set that holds X_j and all variables in $\mathbf{X}$ for which X_j is a parent. Let

$\mathbf{Y'_j} = \mathbf{X} - \mathbf{Y_j}$. Equation 2.7 allows us to rewrite Equation 2.8 as:

$$P(X_i|\mathbf{e}) = \alpha \sum_{\mathbf{X'}-\{X_j\}} \prod_{X_m \in \mathbf{Y_j}'} P(X_m|\Pi_m, \mathbf{e}) \sum_{X_j} \prod_{X_k \in \mathbf{Y_j}} P(X_k|\Pi_k, \mathbf{e}) \tag{2.9}$$

This process can be done for each variable, that is, multiply all of the distributions defined over X_j, marginalize X_j from the distribution, and return it to the pool of distributions. This is exactly the basis of the VE algorithm (discussed in later sections). The complexity of the algorithm is linear on the size of the largest intermediate distribution (the distribution created as a result of marginalizing a single variable).

As an example, suppose a user is interested in the posterior probability distribution over the variable *X-ray* (X), given no evidence. This can be written as follows:[1]

$$P(X) = \sum_{\mathbf{X}-\{X\}} P(V)P(S)P(T|V)P(L|S)P(B|S)P(C|T,L)P(X|C)P(D|C,B) \tag{2.10}$$

where $\mathbf{X}$ represents the union of all the variables in the Asia problem. Rather than performing an entire recombination, V can first be marginalized out by combining all of the factors that include V in their definition:

$$P(X) = \sum_{\mathbf{X}-\{X,V\}} P(D|C,B)P(X|C)P(S)P(B|S)P(L|S)P(C|T,L) \sum_{V} P(V)P(T|V) \tag{2.11}$$

If the variable elimination ordering $\{V, T, L, S, B, C, D\}$ is used, then the above equation can be rewritten as:

$$P(X) = \sum_{D} \sum_{C} P(X|C) \sum_{B} P(D|C,B) \sum_{S} P(S)P(B|S) \sum_{L} P(L|S) \sum_{T} P(C|T,L) \sum_{V} P(V)P(T|V) \tag{2.12}$$

Expanding this out, and denoting by f_Y the intermediate distribution created from marginalizing Y, the size requirements of the intermediate distributions become

[1] We exclude α from these equations, for space consideration, and because no normalization is required when there is no evidence.

clear:

$$
\begin{aligned}
P(X) &= \sum_D \sum_C P(X|C) \sum_B P(D|C,B) \sum_S P(S)P(B|S) \sum_L P(L|S) \sum_T P(C|T,L) \sum_V P(V)P(T|V) \\
&= \sum_D \sum_C P(X|C) \sum_B P(D|C,B) \sum_S P(S)P(B|S) \sum_L P(L|S) \sum_T P(C|T,L) f_V(T) \\
&= \sum_D \sum_C P(X|C) \sum_B P(D|C,B) \sum_S P(S)P(B|S) \sum_L P(L|S) f_T(C,L) \\
&= \sum_D \sum_C P(X|C) \sum_B P(D|C,B) \sum_S P(S)P(B|S) f_L(S,C) \\
&= \sum_D \sum_C P(X|C) \sum_B P(D|C,B) f_S(B,C) \\
&= \sum_D \sum_C P(X|C) f_B(D,C) \\
&= \sum_D f_C(X,D) \\
&= f_D(X)
\end{aligned}
$$

Notice that no intermediate distribution computed over during this process contained more than 3 variables. The efficiency of this process is dependent on the order in which the variables are selected to be marginalized. For instance, suppose the following ordering was chosen: $\{C, D, V, T, L, S, B\}$. The equation becomes:

$$
P(X) = \sum_B \sum_S P(S)P(B|S) \sum_L P(L|S) \sum_T \sum_V P(V)P(T|V) \sum_D \sum_C P(X|C)P(D|C,B)P(C|T,L) \tag{2.13}
$$

The first summation takes place over a distribution of six variables. Choosing an optimal variable ordering for this process is NP-hard [38], however, several heuristics exist that give good orderings in polynomial time [38, 56]. We examine the effect of variable orderings on the complexity of inference more closely in Chapter 4.

It is not always the case that we need to compute over the entire network. Circumstances exist when variables and their associated distributions do not contribute anything to the query value. Two classes of such variables are *barren variable* and *d-separated variables*. We consider each in turn, and show how they can affect the size of the effective network (the subgraph of the original Bayesian network over which we need to compute).

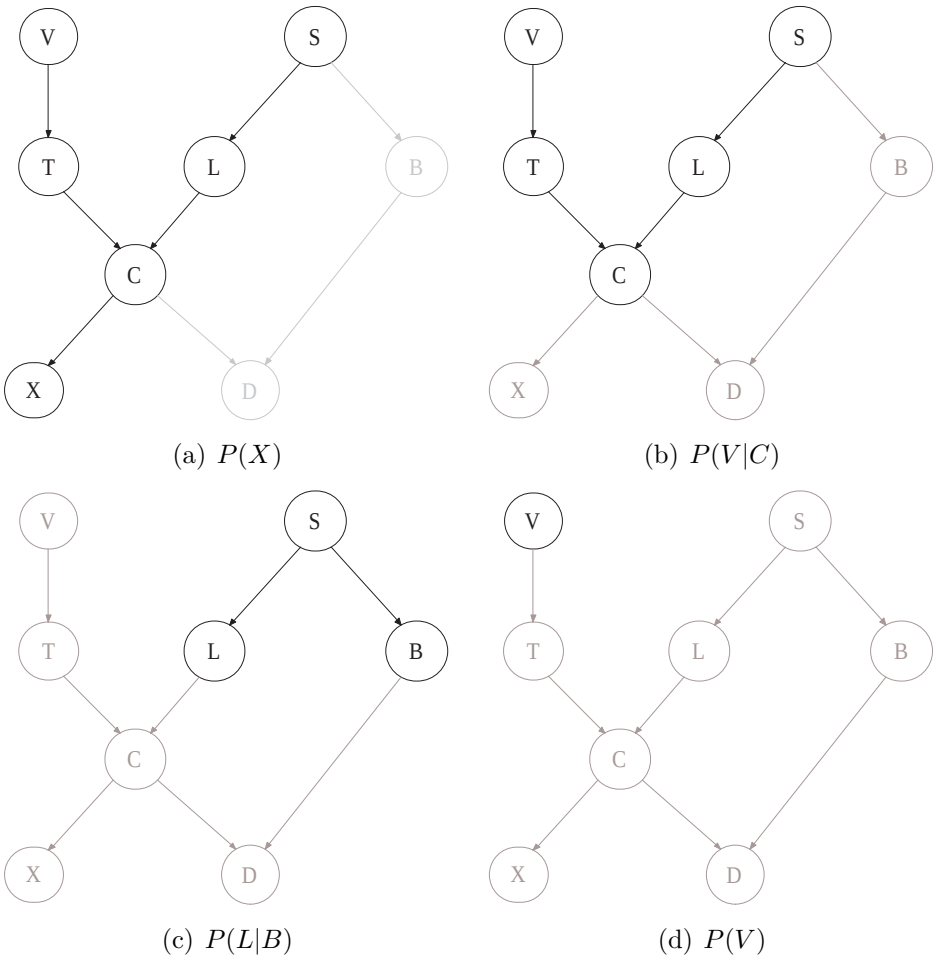
A *barren variable* [59, 60] is a variable that is not part of the query or evidence,

and is either childless or has all barren descendents. Consider the *Asia* network, and recall our previous query, $P(X)$. *Dyspnea* (D) qualifies as a barren variable, since it is not part of the query, not observed, and has no descendents. *Bronchitis* is also barren, as it is not a query or observed node, and its only descendent is barren. Barren variables are computationally irrelevant to probability computations in Bayesian networks, and thus can be excluded. To see this, first consider that if a node X_i is not a query or observed node, then it must be marginalized out. Furthermore, if X_i has no descendents, then it is only defined in one CPT, namely $P(X_i|\Pi_i)$. Hence, marginalization of this variable gives $\sum_{X_i} P(X_i|\Pi_i) = 1$. If X_i has descendents, and they are all barren, then by marginalizing its descendents first, we end up with the same situation (X_i defined only in its own distribution).

Pruning barren variables from the network requires linear time in the number of variables, and can sometimes lead to a substantial decrease in the size of the network. Figure 2.2 shows the Asia network given different queries. Barren variables often comprise a considerable portion of the Bayesian network, especially when the observations and queries are localized to a particular section of the network, and even more so when those observations/queries are shallow (closer to the root than the leaves).

When a variable is part of the evidence, we say that it is *observed*. Observing a variable instantiates it to a particular value, and has the effect of removing its outgoing arcs in the network. Recall that an arc in a Bayesian network indicates that the parent is a conditioning variable in the child's local distribution. Hence, if an arc exists from X_i to X_j, then $X_i \in \pi_j$. Observing event $X_i = x_i$ creates a new distribution in which X_i is not a part of the domain. Since the distribution is no longer defined over X_i, the arc between X_i and X_j can be considered to be removed.

These pruned arcs become important if the network gets separated into distinct parts. If the Bayesian network becomes disconnected into subgraphs, then no two distributions from different subgraphs are defined over a common variable. A query variable X_q is therefore probabilistically dependent only on the subgraph that contains it. To see this, let a Bayesian network over variables $\mathbf{X}$ be divided into two



(a) $P(X)$ (b) $P(V|C)$

(c) $P(L|B)$ (d) $P(V)$

Figure 2.2: Examples of queries and their corresponding relevant networks, where barren variables have been greyed out.

subgraphs: $\mathcal{G}_q$ containing variables $\mathbf{X_q}$ (where $X_q \in \mathbf{X_q}$), and $\mathcal{G}_{\bar{q}}$ containing variables $\mathbf{X_{\bar{q}}} = \mathbf{X} - \mathbf{X_q}$. If the above assumptions hold, then we can write the probability equation of X_q as follows:

$$P(X_q|\mathbf{e}) = \alpha \sum_{\mathbf{X_q}-\{X_q\}} \prod_{X_i \in \mathbf{X_q}} P(X_i|\pi_i, \mathbf{e}) \sum_{\mathbf{X_{\bar{q}}}} \prod_{X_j \in \mathbf{X_{\bar{q}}}} P(X_j|\pi_j, \mathbf{e}) \tag{2.14}$$

The second factor in the product, $\sum_{\mathbf{X_{\bar{q}}}} \prod_{X_j \in \mathbf{X_{\bar{q}}}} P(X_j|\pi_j, \mathbf{e})$, reduces to a constant value, since all of its variables are marginalized. This constant term is found in the denominator of the normalization constant as well. Therefore, the two cancel out, so the second term need not be calculated.

A variable that is *d-separated* from the query variable exists in a subgraph that is disconnected from the subgraph containing the query variable (once barren variables and the arcs from observed variables are pruned). As demonstrated above, this means that the posterior probability distribution over the query is probabilistically independent of a variable that it is d-separated from, and thus such a variable can be ignored during computation.

To formally define d-separation, we take the approach of Shachter[2], and first define an *active* path in the Bayesian network [61]:

Definition 2.1.1. *Let X and Y be two nodes in a DAG, and $\mathbf{Z}$ be a set of nodes from the same DAG. An* active path *from X to Y given $\mathbf{Z}$ is a path such that:*
(1) every node on the path with converging arrows is in $\mathbf{Z}$ or has a descendent in $\mathbf{Z}$
(2) every other node on the path is outside $\mathbf{Z}$

The formal definition of d-separation is as follows [50, 61]:

Definition 2.1.2. *Let $\mathbf{X}, \mathbf{Y}$, and $\mathbf{Z}$ represent three disjoint set of nodes in a DAG.* $\mathbf{Z}$ d-separates $\mathbf{X}$ *and* $\mathbf{Y}$ *if there is no active path between $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ given $\mathbf{Z}$.*

From our example, the set $\{L, B\}$ d-separates $\{S\}$ from $\{X, D, C, T, V\}$. As well, the set $\{\}$ d-separates $\{V, T\}$ from $\{L, S, B\}$.

[2]The original d-separation definition, given by Pearl [50] defines d-separation using negation, and is less intuitive than Shachter's definition.

There have been many algorithms designed to compute posterior probability distributions over variables in the network. Inference in Bayesian networks has been shown to be NP-hard [12]. However, the mechanics of the algorithms allow them to perform well in many cases, reserving exponential behaviour for a specific subset of networks that produce worst-case complexity. These algorithms are the topic for the remainder of this chapter.

There are two basic types of inference algorithms for Bayesian networks. First, *query-based* algorithms compute the posterior probability of a query variable (or set of variables). Hence, the algorithm must be executed once for each query, even if the evidence does not change. The other type of algorithm, which we'll refer to as *batch updating* algorithms, compute the posterior probability of all variables simultaneously. The model presented in this document (beginning in Chapter 3) is an example of a query-based algorithm. We will present both types of algorithms in this chapter, for comparison purposes and completeness.

2.2 Common Methods for Inference

While many techniques have been proposed for calculating probabilities from a Bayesian network, two classes of algorithms are the most popular at the time of writing. Junction-tree propagation methods [35, 36, 40] offer efficient techniques for computing multiple queries simultaneously from the network (at the expense of space and precompilation), while variable elimination [23, 65, 66] calculate only one distribution, but can exploit query-specific independence. Together, these algorithms offer flexibility (the best algorithm can be chosen based on the type of application), as long as the user has sufficient memory to store intermediate distributions.

2.2.1 Variable Elimination

Variable elimination (VE) is a query-based algorithm that formalizes the process used to derive $P(X)$ from the *Asia* network in the previous section. VE computes a distribution over its query variables by marginalizing other variables from the joint

probability one by one.

Variable elimination begins by creating a pool of distributions, which initially contains the CPTs of the Bayesian network. A variable to be marginalized is selected, and all distributions defined over that variable are removed from the pool. These distributions are multiplied into a single distribution, and the selected variable is marginalized from the resulting distribution. This distribution is then placed in the pool, and the process is repeated, until all non-query variables have been marginalized. The remaining distributions in the pool are combined using multiplication, and the resulting distribution is normalized, giving us the posterior probability over the query variables.

The complexity of the algorithm is $\mathbf{O}(n \exp(w_\rho))$, where n is the number of variables in the Bayesian network, ρ is the ordering in which the variables are eliminated, and w_p is the *induced width* of the variable ordering, which is equivalent to the number of variables in the largest intermediate distribution. The induced width is a function of the variable ordering. Finding an optimal elimination ordering is a hard problem, but with several heuristics giving good approximations to the optimal [38, 56].

The primary advantages of the VE algorithm are its simplicity and its dynamic nature. The algorithm is very straightforward to implement, and no precompilation takes place, allowing the algorithm to exploit barren and d-separated variables at runtime. The main disadvantage to VE is that it requires k runs to compute the individual posterior for k variables. Much of the work is repeated for each computation, something that other methods are able to avoid (see Section 2.2.2).

There exist several variants to the VE algorithm. Bucket Elimination [20] places the distributions into separate pools (or *buckets*) according to the domains of the distributions, thus eliminating the need to search for distributions defined over a particular variable when marginalizing. Mini-buckets [22, 25] is an algorithm that computes an approximation to the posterior of the query variables in less time and space than VE. In the mini-bucket algorithm, the set of distributions of a bucket is partitioned into smaller buckets, and each smaller bucket is processed the same as a standard bucket in Bucket Elimination. This further partitioning typically creates

smaller intermediate distributions, which reduces the time and space requirements of the algorithm, at the expense of an exact answer.

2.2.2 Junction-Tree Propagation

Junction-tree propagation (JTP) [35, 36, 40] is a batch update technique that precompiles the Bayesian network into a *junction-tree*. Computing over the junction tree allows the posterior probability of each variable to be computed simultaneously and efficiently.

A junction-tree is an undirected, acyclic graph derived from the Bayesian network. Each node in the junction-tree, called a *cluster*, is a subset of the variables from the Bayesian network. The JTP algorithm calculates a joint probability distribution over each cluster in the junction-tree. Once JTP completes, the posterior probability of a variable can be obtained from any cluster containing that variable by marginalizing out all other variables in that cluster, and normalizing the resulting distribution.

The clusters of a junction-tree are identified after the Bayesian network is *moralized* and *triangulated*. To moralize the Bayesian network, the parents of each variable are *married* (an edge is placed between any two variables that share a common child and do not already have an edge between them), and the direction of all links are dropped (Figure 2.3). Triangulating a graph ensures that any cycles of length greater than 3 have a *chord* intersecting them (Figure 2.4). Triangulating a graph is typically done through an elimination procedure (similar to the algorithms of Section 2.2.1), where one variable is eliminated from the graph, and edges are added between the remaining neighbours of the eliminated variable. The triangulated graph is the original graph with these new added edges. Each maximal clique in the moralized, triangulated graph contains the variables for a cluster in the junction-tree.

Once the clusters of the graph have been identified, the junction-tree can be constructed. A vertex is created for each cluster, and edges are added between the vertices such that 1) the graph is connected, with no loops and 2) the *running intersection* property is maintained. If a junction-tree maintains the running intersection property, then if two clusters share a common variable, then all clusters along the

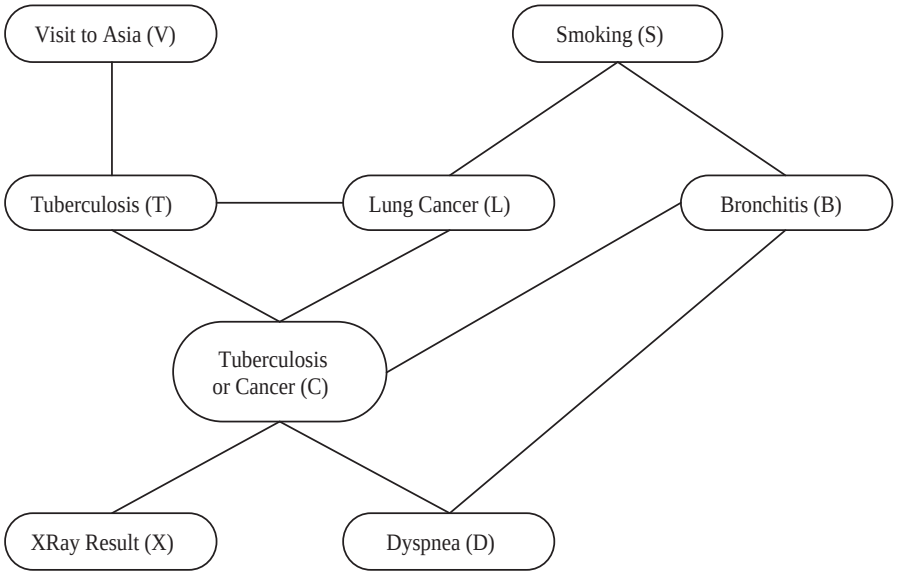


Figure 2.3: The *Asia* network after moralization. Note the marriage between T and L, and between C and B. As well, the direction of the arcs has been dropped.

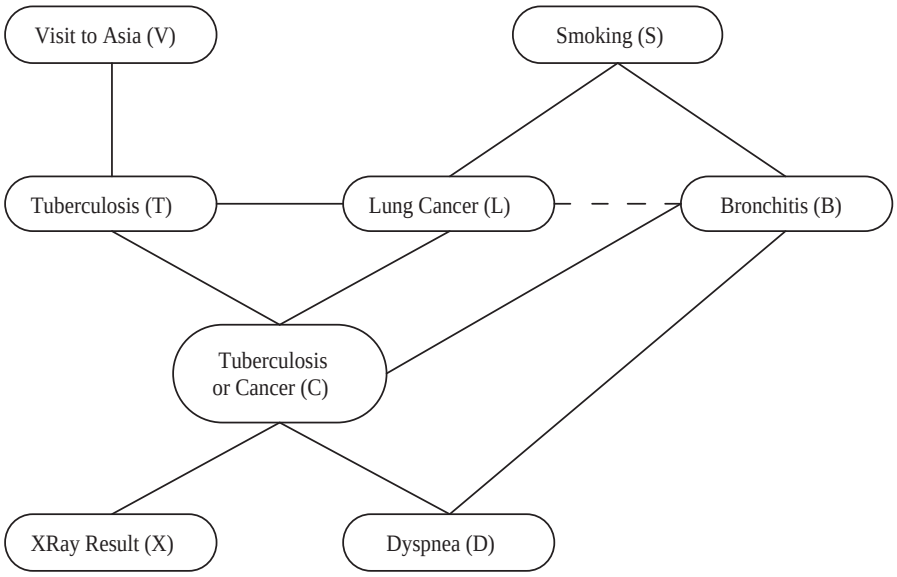


Figure 2.4: The *Asia* network, triangulated.

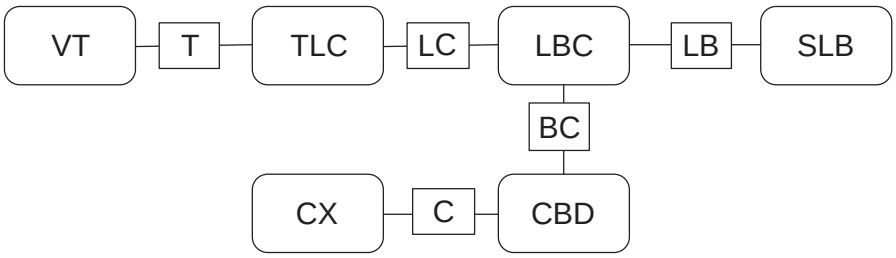


Figure 2.5: A junction-tree for the *Asia* network. Clusters are shown as rectangles with rounded corners, and separator sets are shown as rectangles with square corners.

path between those two clusters contain the variable as well. Each edge in the junction tree is also labeled with a variable set, known as its *separator set*. The separator set is just the intersection of the clusters that the edge connects. Figure 2.5 shows a junction-tree for the *Asia* network.

Inference in a junction tree proceeds with each nodes passing messages to each other. These messages take the form of a distribution. One message is passed from each cluster to each of its neighbours. These messages are combined into a final distribution at each node, and the posterior probability for a variable at a cluster can be obtained by marginalizing away all other variables in the cluster.

The complexity of inference in junction-trees is $\mathbf{O}(n \exp(w))$, where w is the the size of the largest clique. The clique sizes depend on the triangulation of the Bayesian network, which in turn depends on the variable ordering used to determine fill-edges. Finding the optimal variable elimination ordering for this problem is NP-hard [38]. In fact, the problem of finding an optimal variable ordering is the same for both VE and JTP, so the same heuristics can be applied.

The primary advantage of JTP is that it calculates the individual posterior of each variable simultaneously. That is, after the completion of the algorithm, the posterior of any variable is available from the distribution of any cluster containing that variable. One disadvantage of a junction-tree is that its space requirement is exponential on the size of its largest clique. As well, because the junction-tree is a precompiled structure, it is more difficult to take advantage of barren variables and d-separation.

At the time of writing, junction-tree algorithms are the most popular algorithms for inference in Bayesian networks. They are prevalent in commercial systems (Netica [47], Hugin [6]), and have extensive research behind them for well over a decade. Algorithms exist to optimize structure [28], handle evidence dynamically [13, 27], and run query-driven inference (for generating beliefs over small subsets of variables; this requires that functions only be stored over the separators) [21, 34]. For more indepth analysis of junction-tree algorithms, including implementations, the reader is encouraged to consult Huang and Darwiche [34].

2.3 Conditioning Algorithms

While inference in Bayesian networks requires exponential time, it does not always have to require exponential space. Conditioning algorithms provide a space-efficient alternative to the algorithms of the last section. Conditioning algorithms transform inference into smaller subproblems, and then recombine the solutions to these subproblems into the overall solution.

As with the popular methods, conditioning methods can be classified into batch updating and query-based algorithms. The former transform the network to a polytree (singly connected), whereas the latter is a divide and conquer approach to inference. The former will be the topic of this section; the latter will be considered in the next section.

2.3.1 Global Calculation

Conditioning algorithms that update all posterior probabilities (batch update) follow the same general format: choose a cutset $\mathbf{C}$ whose instantiation renders the network *singly connected* (no directed or undirected loops), recalling that instantiating a variable to a value prunes its outgoing arcs. When the graph is singly connected, the probabilities over the current context can be calculated using Pearl's message passing algorithm. This must be done once for each context of the cutset. The details of this are considered later; the message passing algorithm is introduced first.

Note that the message passing algorithm is presented here because of its relationship to conditioning methods; the algorithm itself is not a conditioning algorithm.

Message Passing

Pearl's message passing algorithm [49, 50] computes the posterior probability distribution of each variable in a Bayesian network in a single run. The algorithm only computes correct probabilities for a singly-connected network. Hence, the algorithm is typically used in conjunction with a conditioning algorithm that renders the network singly-connected.

During the message passing algorithm, a variable in the Bayesian network becomes a processing unit. The variable receives *messages* from its neighbouring nodes. These messages are in the form of a distribution, representing information from another part of the network. A variable uses these messages to calculate the posterior probability distribution over itself, as well as to calculate messages to send to its neighbours. The message sent to a neighbouring variable is a summary of all information received from all other neighbours. The algorithm terminates when all messages have been sent.

The number of messages sent during the message passing algorithm is $2e$, where e is the number of arcs in the network (since a node sends and receives one message from each neighbour). Calculating messages to be sent to parent variables takes $\mathbf{O}(\exp(f))$ time, where f is the size of the largest family (calculating a message to be sent to a child can be done in time linear on the size of the variable, once the posterior probability has been calculated, and therefore does not contribute to the complexity). Calculating the posterior probability of a variable from the messages also takes $\mathbf{O}(\exp(f))$ time. Hence, the overall time for the algorithm is $\mathbf{O}((n + e)\exp(f))$. The space required by the algorithm is $\mathbf{O}(n\exp(f))$, for CPT storage (the messages passed are linear in the domain size of the variables, and therefore do not contribute to the space complexity).

The advantages of Pearl's algorithm is its low resource requirements: in terms of complexity, it is among the fastest and smallest inference algorithms for Bayesian

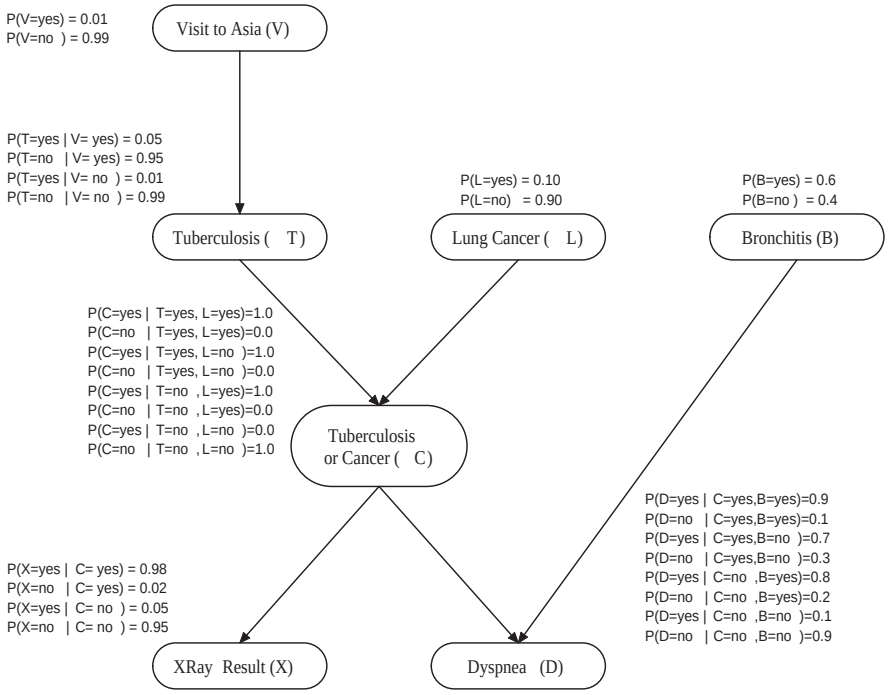
networks to date. The algorithm calculates posterior probability distributions for each variable simultaneously, as opposed to a single distribution as in VE. Also, because each variable processes independently, much of the computation can be done in parallel. However, the algorithm works only for singly-connected networks, which in practice occurs infrequently.

Pearl conjectured that running the message passing algorithm in a multiply-connected network (containing undirected loops) might stabilize to an equilibrium, even though the posteriors at equilibrium may not be representative of the real posteriors. Murphy et. al [45] explored this idea on general probabilistic networks, attempting to ascertain empirically if message-passing was a reasonable approximation approach on "loopy" networks. The results showed that when convergence occurred, the approximations were quite good, outperforming other standard approximation methods given a similar amount of running time. However, the algorithm would exhibit oscillatory behaviour over certain networks, and never converge. The oscillation seemed to have correlation to small prior probabilities (the authors were able to correct oscillation in some networks by increasing some of the prior probabilities).

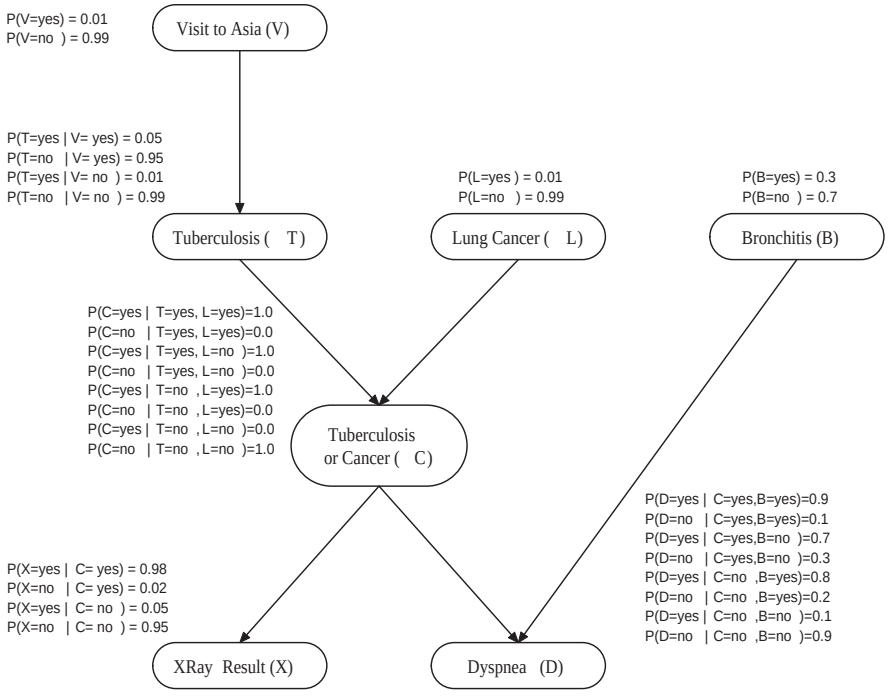
Cutset Conditioning

Cutset conditioning [50] is an inference algorithm for Bayesian networks that uses the message-passing algorithms as its probability calculator. The central idea behind cutset conditioning is to choose a *cutset*, or set of variables from the Bayesian network whose instantiation renders the network singly-connected. Recall that when a variable is instantiated, it's outgoing arcs are pruned from the network. Consider the Asia example. Message passing cannot be applied to this network, as it is not singly-connected. However, by instantiating *Smoking*, we break the only loop in the graph (shown in Figure 2.6), and we may now apply the message passing algorithm.

Given such a cutset $\mathbf{C}$, instantiating the variables to $\mathbf{C} = \mathbf{c}$ and applying Pearl's algorithm calculates $P(X_i|\mathbf{c}, \mathbf{e})$ for each X_i in the network. To obtain the desired posterior probability distribution $P(X_i|\mathbf{e})$, we can use the law of total probabilities:



(a) The Asia network, conditioned on $S = true$.



(b) The Asia network, conditioned on $S = false$.

Figure 2.6: The Asia network, conditioned on S. Notice that in each case, the network is singly connected, and the beliefs can be updated using message passing.

$$P(X_i|\mathbf{e}) = \sum_{\mathbf{c}\in\mathbf{C}} P(X_i|\mathbf{e},\mathbf{c})P(\mathbf{c}|\mathbf{e}) \tag{2.15}$$

In other words, the message-passing algorithm is used once for each instantiation of the cutset. Calculating $P(\mathbf{c}|\mathbf{e})$ is actually calculated as $\alpha P(\mathbf{e}|\mathbf{c})P(\mathbf{c})$, and both terms in this equation can be calculated using message passing.

Let $|\mathbf{C}|$ be the number of variables in the cutset. The time complexity of the algorithm is $\mathbf{O}(\ [(n+e)\exp(f)]^{|\mathbf{C}|}\ )$. Finding the optimal (smallest) cutset is NP-hard [63]; different methods have been suggested for finding such a cutset [8,26,62,63]. The space complexity of the algorithm is $\mathbf{O}(n\exp(f))$, since it requires enough space to run the message-passing algorithm.

The primary advantage of conditioning is its low memory requirements. Since the largest family of the Bayesian network is almost always smaller than its induced width, the conditioning algorithm can achieve an exponential space saving over JTP and VE. However, the size of the cutsets in general Bayesian networks tend to give these algorithms longer runtimes than JTP and VE. The difference in runtime can be substantial, and conditioning algorithms do not enjoy the same popularity that JTP and VE have.

Peot and Shachter [51] improve on the original cutset conditioning algorithm by defining multiple cutsets per network - one for each *knot*. A knot is a subgraph of the network that cannot be disconnected by removing one edge. This allows conditioning only over relevant variables, rather than conditioning every variable over the entire cutset. In the worst case, the knot conditioning algorithm has the same complexity as standard conditioning, but is often much better. Local conditioning [26] is a further refinement of knot-conditioning. In local conditioning, conditioning is applied exclusively within each loop. Local conditioning provides a depth-first search algorithm for detecting the cutsets of each loop. In practice, knot-conditioning is as least as good as global conditioning (often much better), while local conditioning is as least as good as knot-conditioning (and often much better).[3] Dechter [21] intro-

[3]The authors report linear to exponential ratios between local and knot conditioning in some

duced a hybrid approach that uses a version of JTP with conditioning. The result is a time-space tradeoff: the algorithm works in space-constrained environments, and the runtime is inversely proportional to the amount of memory available. Bounded conditioning [32] is an algorithm that uses conditioning to approximate bounds on posteriors. Probabilities that have not yet been calculated are replaced with the interval $[0, 1]$ in Equation 2.15, giving upper and lower bounds on the final posteriors. As the actual probabilities values are calculated exactly, they replace the intervals in the equation. Hence, as time progresses, the bounds get better and better.

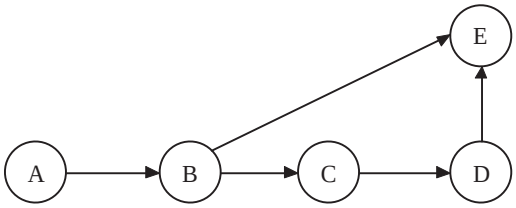
2.4 Divide and Conquer Conditioning

Divide and conquer conditioning [15, 44, 55] is similar to cutset conditioning, in that it uses a *cutset* to condition the network over a specific context. However, rather than this cutset being chosen to make the network singly-connected, the cutset *partitions* the network into two d-separated components. The partitions are solved in a recursive manner, and the results are recombined to obtain the solution.

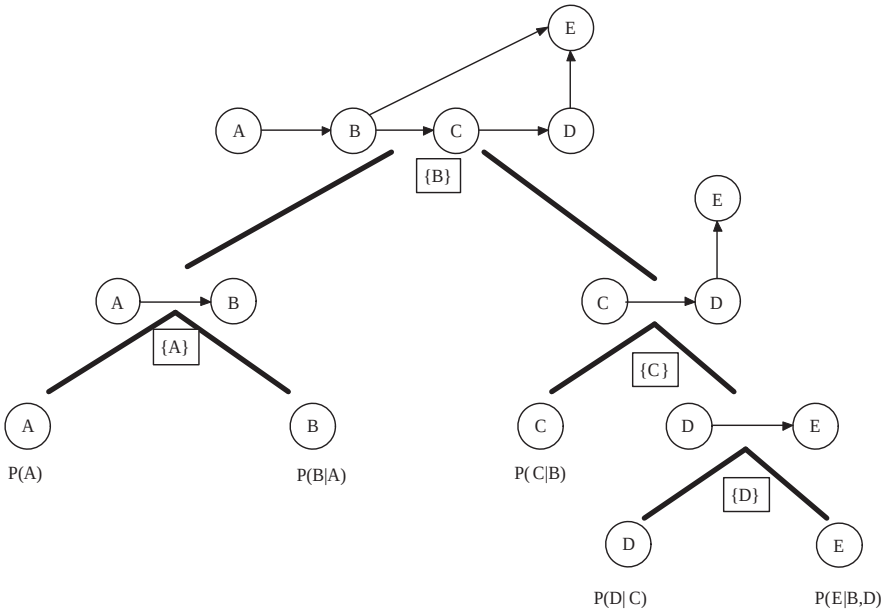
Divide and conquer conditioning begins by recursively partitioning the Bayesian network into a structure called a *dtree*. Figure 2.7 shows a Bayesian network, and a dtree compilation of the network (taken from Darwiche [15]). Each internal node in the dtree represents a subgraph of the original Bayesian network. Each internal node N also contains a set of variables known as its *cutset*. The cutset at a particular node d-separates its subgraph into two distinct subgraphs: these subgraphs become the children of that node. The leaves represent single variables of the network (a single variable cannot be further partitioned). The subgraph of an internal node is implicit: each internal node stores only the cutset, while the leaves store the CPT associated with its labeling variable.

Once construction of the dtree is complete, it is used to calculate the probability of a context $P(\mathbf{C} = \mathbf{c})$, where $\mathbf{C}$ is a subset of the nodes in the Bayesian network. Given a dtree node T, let T^l and T^r represent the left and right children of T,

examples.



(a) An example Bayesian network.



(b) A recursive decomposition of the network.

Figure 2.7: An example Bayesian network (from Darwiche [15]) and a recursive decomposition of that network. The cutsets at each node are shown in each box.

respectively, and let $cutset(T)$ represent the cutset at T. The value calculated from T given $\mathbf{c}$, denoted $P_T(\mathbf{c})$, is as follows:

$$P_T(\mathbf{c}) = \sum_{\mathbf{d} \in \mathcal{D}(cutset(T))} P_{T^l}(\mathbf{c} \wedge \mathbf{d}) P_{T^r}(\mathbf{c} \wedge \mathbf{d}) \tag{2.16}$$

that is, the value is calculated by recursively calculating the value at T^l and T^r for each $\mathbf{d} \in \mathcal{D}(cutset(T))$. When T is a leaf node, the context passed to this node contains an assignment to each variable in the domain of the CPT at T; the value returned from T is the value in this CPT that corresponds to the context.

As mentioned, the probability calculated from a dtree is the probability of a context $\mathbf{C} = \mathbf{c}$, not a posterior probability. To calculate a posterior probability distribution $P(X_i|\mathbf{e})$ from a dtree, $P(x_i \wedge \mathbf{e})$ is calculated for each $x_i \in \mathcal{D}(X_i)$, and the resulting vector is normalized.

There are several variations of divide and conquer conditioning. Recursive conditioning [15] and recursive decomposition [11, 44] decompose the network into the described binary tree structure (a dtree). Adaptive conditioning [55] is an adaptation of recursive conditioning that allows one to tailor a query both by time and space requirements. The algorithm differs from the other decompositions in that it does not attempt to decompose the network to single-nodes, and the decomposition is not necessarily binary. Instead, it decomposes based on memory requirements, and may choose to run an inference algorithm on a multi-node subnetwork. A network is only decomposed further if memory requirements do not suffice to run inference on the current decomposition.

The time complexity of recursive conditioning is $\mathbf{O}(n \exp(w_c d))$, where w_c is the size of the largest cutset, and d is the depth of the tree [15]. To see this, let N be a node in a dtree. Its *a-cutset* is defined as the union of all cutsets in its ancestors in the dtree. During the recursive conditioning algorithm, N will be called once for each instantiation of its a-cutset, which in the worst case is of size $w_c d$. If a dtree is constructed using an elimination ordering, then it can be shown that the largest cutset will be bounded from above by the induced width of the variable ordering. As

well, the tree can be balanced using rake and compress methods [43], such that it has logarithmic height while only affecting w_c by a constant factor. Hence, the resulting time complexity of the algorithm is $\mathbf{O}(n \exp(w \log n))$, where w is the width of the variable ordering used to construct the dtree, and n is the number of variables in the network.

The memory requirements of recursive conditioning are much smaller than JTP and VE. The CPT storage requires $\mathbf{O}(n \exp(f))$ storage, where f is the size of the largest family. Excluding the CPTs, the space required to store the dtree structure is linear on the number of nodes in the network. During computation of $P(\mathbf{E} = \mathbf{e})$, the algoritm traverses the tree structure in a depth-first fashion, storing only the current recursive path, which is linear on the height of the tree. Hence, even though the space complexity of recursive conditioning is asymptotically the same as other conditioning algorithms, the actual amount of space required by the algorithm is typically less. This small memory requirement is the primary advantage of recursive conditioning.

2.4.1 Caching in Recursive Decompositions

Recursive decompositions require more time to compute over than JTP and VE. The extra time complexity is due to repeat computation, which is illustrated in the following example. Figure 2.8 shows the *Asia* network compiled into a dtree. Recall that a node will be called once for each instantiation of its a-cutset. Consider the node labeled with the cutset $\{V\}$ in the graph. This node will be called once for each instantiation of its a-cutset, which is $\{L, B, T\}$. Table 2.4.1 shows the results of each of these calls to the node. Each entry in the table shows the current context of the a-cutset, and the return value of each call to the node. Notice that the return value is the same for all contexts when $T = yes$, as well as for $T = no$. Hence, the algorithm recalculates these values unnecessarily.

Recomputation can be avoided by storing these values once they are calculated, a technique known as *caching* [15]. If the value $\sum_{v \in V} P(v)P(T = yes|v)$ were stored

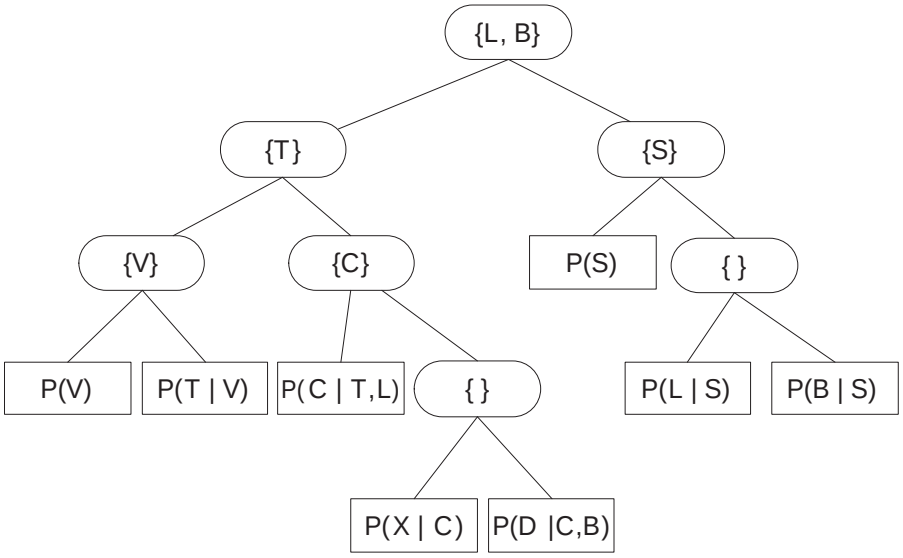


Figure 2.8: The *Asia* network, compiled into a dtree.

Table 2.1: Trace of visits to node labeled with $\{V\}$ in Figure 2.8.

Visit	**Context**	**Value**
1	$L = yes, B = yes, T = yes$	$\sum_{v \in V} P(v)P(T = yes\|v)$
2	$L = yes, B = yes, T = no$	$\sum_{v \in V} P(v)P(T = no\|v)$
3	$L = yes, B = no, T = yes$	$\sum_{v \in V} P(v)P(T = yes\|v)$
4	$L = yes, B = no, T = no$	$\sum_{v \in V} P(v)P(T = no\|v)$
5	$L = no, B = yes, T = yes$	$\sum_{v \in V} P(v)P(T = yes\|v)$
6	$L = no, B = yes, T = no$	$\sum_{v \in V} P(v)P(T = no\|v)$
7	$L = no, B = no, T = yes$	$\sum_{v \in V} P(v)P(T = yes\|v)$
8	$L = no, B = no, T = no$	$\sum_{v \in V} P(v)P(T = no\|v)$

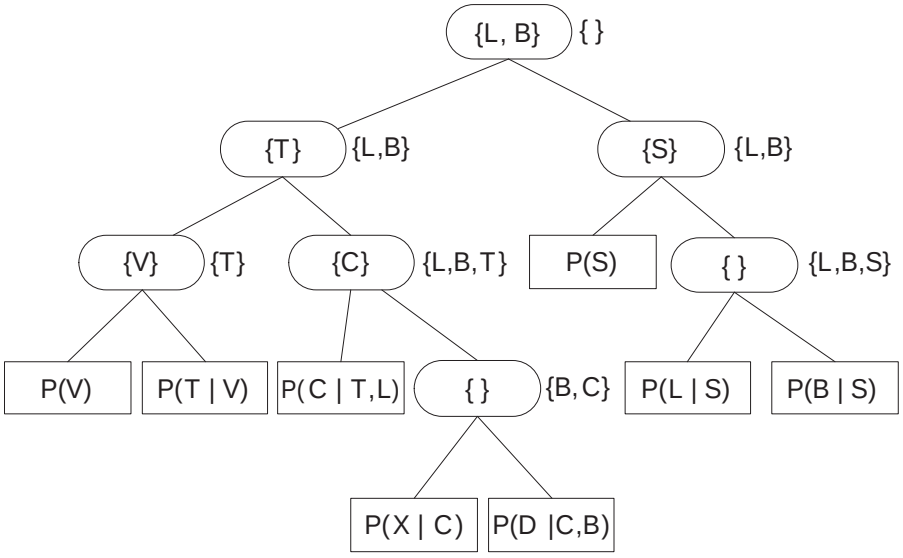


Figure 2.9: The *Asia* dtree, with cache-domains shown to the right of the internal nodes.

after visit 1, and $\sum_{v \in V} P(v)P(T = no|v)$ after visit 2, then all subsequent visits to the node would require a constant time lookup to the cache; the value returned would depend on the current value of T in the context of the a-cutset.

Formally, define the *cache-domain* of a node N, denoted $CD(N)$, as the intersection of N's a-cutset and the union of all variables in the CPTs of its leaf variables.[4] The values returned from N will depend only on the current context of $CD(N)$ in N's a-cutset. In the above example, the cache-domain of the node labeled by $\{V\}$ is $\{T\}$. The recursive conditioning algorithm can be modified as follows: when visiting a node, it first checks the cache at that node to see if a value for the context of the cache-domain has already been calculated. If it has, it simply returns this value. If not, it calculates this value, stores it in cache, and returns it. Figure 2.9 shows the *Asia* dtree, with its cache-domains shown to the right of each internal node.

When caching is employed, the time requirements of recursive conditioning are reduced, while the space requirements are increased. Rather than a node being called once for each instantiation of its a-cutset, it is now called once for each instantiation of its parent-node's cache-domain unioned with its parent-node's cutset. It can be

[4]Darwiche et. al [4, 5, 15] refers to this set as simply a *context*, we use *cache-domain* to avoid confusion with our previous definition of context.

shown that the cache-domain size at each node is bounded by the induced width of the variable ordering used to construct the dtree [15]. This means that the time and space complexity of recursive conditioning with caching is $\mathbf{O}(n \exp(w))$, the same as JTP and VE.

Dead Caches

Caching all possible values increases the space requirements of recursive conditioning to $\mathbf{O}(n \exp(w))$, or the same as JTP and VE. However, Allen and Darwiche [3] demonstrated that the memory requirements of caching can be reduced by identifying *dead caches*. Dead caches in a recursive decomposition are caches whose values would only be calculated and never queried. Dead caches are never allocated any memory, so the overall space requirements for caching is reduced.

As an example, consider the *Asia* dtree in Figure 2.9, in particular the node labeled with $\{C\}$. The cache-domain at this node is $\{L, B, T\}$. The node is visited only once for each instantiation of its cache-domain, therefore, the cache is never queried.

Dead caches can be identified in dtrees as a cache whose context is a superset of its parent's context. These caches can be removed from recursive decompositions with no runtime consequence. The memory savings afforded by dead cache removal can be substantial. Figure 2.10 shows the *Asia* example with dead caches removed (grayed out). In this example, the live caches requires less than 20% of the original space required by full caching.

Allen and Darwiche showed empirically that many of the caches in dtrees constructed from test Bayesian networks were dead caches, and their removal considerably improved the space efficiency of recursive conditioning. They also showed that the memory required was substantially less than both JTP and VE as well, while the time complexity remains the same as for JTP and VE.

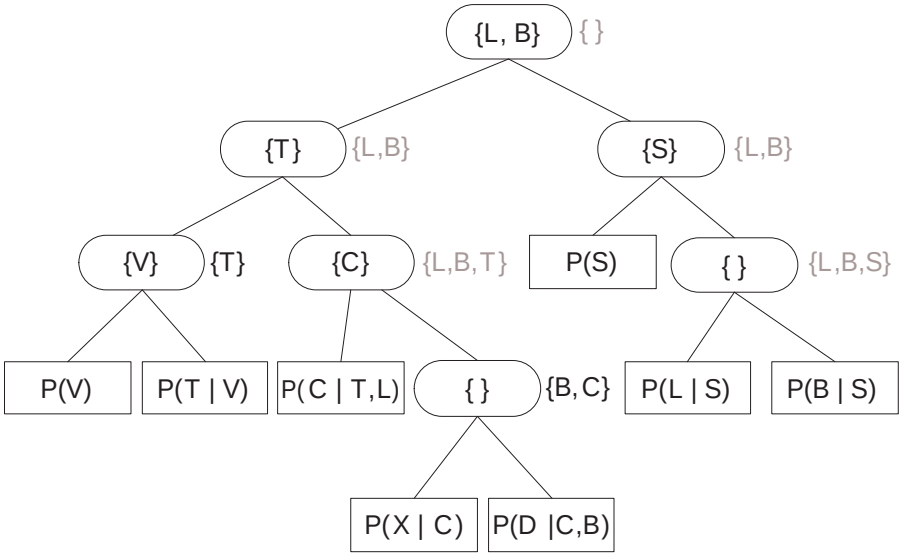


Figure 2.10: The *Asia* dtree, with dead caches grayed out.

Partial Caching

A dtree with cache may require too much memory to store in some applications, even after the removal of dead caches. Darwiche et. al [5, 15] demonstrates *partial caching*, or caching only a subset of the possible values, where the size of the subset is dictated by the amount of available memory. This creates a *time-space tradeoff*, where increasing the amount of memory decreases the required computation time, and vice versa. These time-space tradeoffs are very useful, especially considering that they can be calculated before any probability computations take place. Given a fixed amount of memory, the tradeoff curves indicate the amount of time each computation will take; likewise, the curves indicate the amount of memory required to compute a probability in a given interval.

Given that a partial caching scheme is chosen, one must choose which values to cache. There are two overlapping systems in the literature:

1. *Discrete vs. non-discrete.* In discrete caching, each node either caches all of its values, or none of its values. Non-discrete caching allows for a subset of the cache values at a node to be stored.

2. *Uniform vs. non-uniform.* In uniform caching, each node receives an equivalent

percentage of its required cache space. In non-uniform caching, the percentage of values a node is allowed to cache is specific to that node.

The non-caching and full-caching models described at the beginning of this section are examples of discrete uniform caching models. Non-discrete non-uniform caching, while probably the most useful model, is also the hardest to optimally calculate, as it involves solving a nonlinear system, and is an open problem. Non-discrete uniform caching and discrete non-uniform caching have both been studied [4, 5, 15], and are considered here in turn.

Non-discrete uniform caching. In non-discrete uniform caching, each node is allowed to store a certain subset of its cacheable values. The ratio of allowed values vs. total values is the same for each node. No restrictions are placed on which of a node's cacheable values are cached; random selection is assumed in Equation 2.17 (below). When a node is queried, only the subset of cached values is checked. If the current context represents an uncacheable value, the value is calculated as if it was not cached.

Uniform caching is typically inferior to non-uniform caching, as caching at some nodes provides better performance improvements than caching at other nodes. Moreover, pure uniform caching is typically impossible, as the number of values cached is always discrete, and the computed cacheable values may result in a fraction (consider a cache of size 4 with a caching factor of 80%). This leads to "overflow" memory which, if allocated to the dtree nodes, creates a recursive problem. However, this caching scheme does have two advantages. First, the percentage of cacheable values a simple ratio of *memory available* to *number of cacheable values*. Hence, the caching scheme is computable in constant time, given a memory size. This is in contrast to non-uniform models, where finding the optimal set of nodes to cache at requires a search. Second, in this type of partial caching model, the number of recursive calls made to a node N is easily approximated. Let N_P represent the parent of node N in a dtree. The number of calls to N can be approximated as follows [15]:

$$Calls(N) = |\mathcal{D}(cutset(N_P))| \times [cf\ |\mathcal{D}(CD(N_P))| + (1 - cf)Calls(N_P)] \qquad (2.17)$$

where cf is the percentage of cache values allowed at each node. This means that an estimate on the total time required for computing over the dtree given a particular memory size can be calculated before the actual computation takes place, in linear time.

Discrete, non-uniform caching. Discrete non-uniform caching is the most popular model in terms of amount of research. The appeal of this model is twofold: it is much more flexible than uniform caching, but more restricted than non-discrete caching, reducing the search space for an optimal configuration exponentially.

In a discrete caching model, when the amount of memory available prevents caching at all nodes, only a subset of the dtree nodes are allowed to cache, and those nodes cache all values. Different choices produce different runtimes (Equation 2.17); the problem is determining the choice that results in the fastest computation (Darwiche and Allen refers to this as the *Cache Allocation Problem* [4, 5]).

To find a good solution to the Cache Allocation Problem, Darwiche and Allen formulate it as a search problem, and use a branch and bound approach to search through all allowable configurations conforming to a particular memory size [4]. However, even with clever heuristics and pruning methods, the size of the search space is too big for moderate to large networks. To alleviate this problem, a non-backtracking greedy search is used, that provides excellent results in comparison to the branch and bound method [5]. The greedy method is quadratic, computing results in seconds that are nearly as good as the branch and bound results computed over an hour.

2.5 Offline Compilation

Inference in Bayesian networks typically has its computation categorized into two types: offline and online computation. Offline computation, or *compilation*, refers to computation general to the Bayesian network. Examples of this include generating good elimination orderings and computing secondary structures (cluster graphs, dtrees). Online computation refers to computation that is context-dependent, such as evidence observation/retraction, and posterior querying.

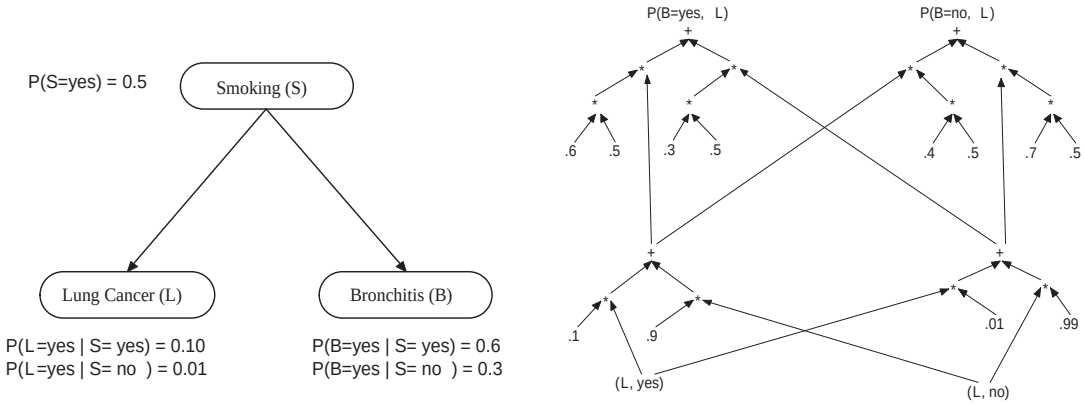
An important distinction between offline and online computation is that programmers are typically only interested in the runtime of the latter: the time cost of offline computation is typically ignored (as the compilation time can be amortized over many runtimes). Such a paradigm is valid considering that the user is typically only interested in the finished product (not the precompiled version). Hence, the more computation that can be done offline, the faster the online computation becomes.

The idea of offline compilation is central to the ideas of this document. We present two previous offline compilation methods in this section: Query-DAGs and Arithmetic circuits.

2.5.1 Query-DAGs

Darwiche and Provan [18] pointed out that the computation of a posterior probability from evidence depends only on the set of evidence variables, and not the specific values of E. That is, while different contexts of the same evidence clearly change the numbers, they do not change the structure of the computation. This idea can be exploited by generalizing the inference operation into an arithmetic expression parameterized by the value of the evidence. The structure that stores this expression is known as a Query-DAG.

A Query-DAG [18, 19] (Q-DAG) is an arithmetic expression in the form of a directed acyclic graph. The roots of the graph represent the operands of the expression, which can be either a number or a special variable called an *Evidence Specific*



(a) Network (b) Q-DAG

Figure 2.11: A portion of the Asia network and an example Q-DAG compilation given the query $P(B|L)$.

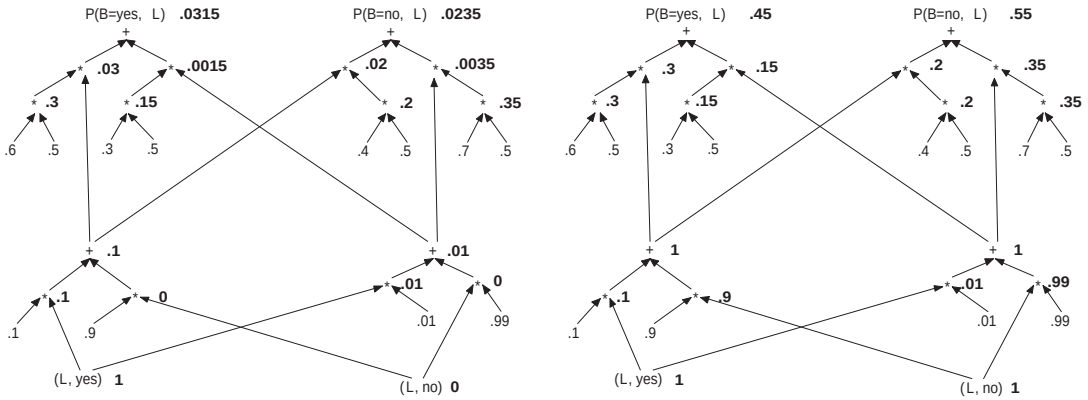
Node (ESN). An ESN corresponds to the instantiation of an evidence value). The non-roots represent arithmetic operations (either * or +), and the leaves of the tree correspond to a query. Figure 2.11 shows a portion of the Asia network and its corresponding Q-DAG, given that our query is $P(B|L)$.

Computation over a Q-DAG is very straightforward, and its evaluation function is very compact. The solution to any query can be found by taking the value of its corresponding leaf node. The value of any operator node can be found by taking the value of its parents, and combining those values with that operator. The value of any number node is simply the number. And the value of an ESN node is 1 if the instantiation represented at that node is consistent with the evidence, and 0 otherwise. Figure 2.12 shows two different instantiations of our Q-DAG, one for $P(B|L = yes)$ and one for $P(B)$. The value of each non-numeric node is shown to the right of the node.

A Q-DAG is essentially an explicit representation of the arithmetic operations that occur during inference. Hence, we can construct a Q-DAG by slightly modifying another inference algorithm. There are three modifications necessary:

1. All values in the CPTs are replaced with a Q-DAG root node containing that value.

2. The indicator vector of any evidence variable is replaced with a vector of



(a) $P(B|L = yes)$ (b) $P(B)$

Figure 2.12: Two instantiations of the Q-DAG from Figure 2.11.

ESNs. From our example, the indicator vector of Lung Cancer would become $\{(L, yes), (L, no)\}$. Note that we can omit any indicator vectors for nodes that are not in the evidence set, which is known a priori.

3. All arithmetic operations of inference are replaced with DAG construction algorithms. Note that because of the first two modifications, the operands to the operators will now be Q-DAG nodes, rather than values. Given an operator in $\{*, +\}$, the operator creates a Q-DAG node labeled with the operator and whose parents are the operands of the operator.

The time complexity to initialize the values at the leaf nodes is the same as the time complexity of the inference algorithm used to construct the Q-DAG. However, once initialized, the Q-DAG can be updated as evidence changes, by propagating the change from the evidence node to its connected leaf nodes. Since only the relevant section of the Q-DAG is updated, this avoids redundant computation, and makes the Q-DAG very time efficient. Because each node in the Q-DAG represents an arithmetic operation from inference, the space complexity is the same as the time complexity of its constructing inference algorithm [18].

The primary advantage of a Q-DAG is that much of its work is performed offline during compilation; such computation typically is not considered part of inference, as it can amortized over many queries. As well, their simple representation and

computation algorithm make them suitable for implementation in primitive architectures. However, Q-DAGs require a large amount of memory, more so than the space required by other inference algorithms, because all intermediate values are stored explicitly.

2.5.2 Arithmetic Circuits

Following his work on Query-DAGs, Darwiche proposed an approach for inference in Bayesian networks based on partial differentiation [16]. The Bayesian network is compiled into a multivariate polynomial. The partial derivatives of this multivariate polynomial can then be used to solve a large class of probabilistic problems in constant time, including posterior probabilities, parameter estimation, model validation, and sensitivity analysis.

To illustrate this, we consider an example similar to the one from Darwiche (but consistent with our previous model). Consider a partial version of the *Asia* network containing only two nodes, *Smoking(S)* and *Lung Cancer(L)*. If we multiply the two CPTs together, we obtain the joint probability over the entire model. Table 2.2(a) shows the result of this multiplication.

Table 2.2: The joint probability distribution and its annotation with evidence indicators.

S	L	$P(S,L)$
yes	yes	.05
yes	no	.45
no	yes	.005
no	no	.495

(a)

S	L	$P(S,L)$
yes	yes	$.05\lambda_s\lambda_l$
yes	no	$.45\lambda_s\lambda_{\bar{l}}$
no	yes	$.005\lambda_{\bar{s}}\lambda_l$
no	no	$.495\lambda_{\bar{s}}\lambda_{\bar{l}}$

(b)

To obtain any distribution $P(x)$, we simply sum together all distributions consistent with x. Hence, $P(s = yes, l = no) = .45$ and $P(l = yes) = 0.055$. Such a selection procedure does not allow for a polynomial-type distribution. However, we can annotate the entries such that they can. For brevity, we will denote $S = yes$ and

$S = no$ as s and $\bar{s}$, respectively. For each variable X in the Bayesian network, and for each value $x \in \mathcal{D}(X)$, define a variable called an *evidence indicator* and denote it λ_x. Hence, the the Bayesian network *Smoking* has two evidence indicators: λ_s for when *Smoking* is true, and $\lambda_{\bar{s}}$, for when smoking is false. These evidence indicators are set to 1 if they are consistent with the evidence, and 0 otherwise (exactly analogous to the ESNs from Q-DAGs). We can annotate each entry in the joint probability distribution with all evidence indicators consistent with its corresponding context. Table 2.2(b) shows this annotation. The network can now be represented by a multivariate polynomial:

$$F(\lambda_s, \lambda_{\bar{s}}, \lambda_s, \lambda_{\bar{s}}) = .05\lambda_s\lambda_l + .45\lambda_s\lambda_{\bar{l}} + .005\lambda_{\bar{s}}\lambda_l + .495\lambda_{\bar{s}}\lambda_{\bar{l}} \tag{2.18}$$

Darwiche further generalizes the algorithm by parameterizing the factors. This allows for the calculation of other problem classes, including parameter estimation, model validation, and sensitivity analysis. Because our focus here is strictly inference, we do not show this parameterization, the interested reader is encouraged to consult the cited literature.

The polynomial contains one monomial for each context of the variables in the network, which means the number of monomials corresponds to the number of entries in the joint probability distribution. This makes such a representation intractable for non-trivial networks. However, this canonical representation of the polynomial can be represented in factored form. The factored form for the polynomial above is:

$$F(\lambda_s, \lambda_{\bar{s}}, \lambda_s, \lambda_{\bar{s}}) = .5\lambda_s(.1\lambda_l + .9\lambda_{\bar{l}}) + .5\lambda_{\bar{s}}(.01\lambda_l + .99\lambda_{\bar{l}}) \tag{2.19}$$

These polynomials are represented in a manner similar to Q-DAGs - as a rooted graph. However, the variables in this case represent the leaves, rather than the root. The graph is referred to as an *Arithmetic Circuit* (AC). Figure 2.13 shows the above polynomial as an AC. The root of the tree represents the function $F(\mathbf{e})$, where the $\mathbf{e}$ vector is used to instantiate the evidence indicators.

Once in polynomial form, the posterior probability of an event X given evidence

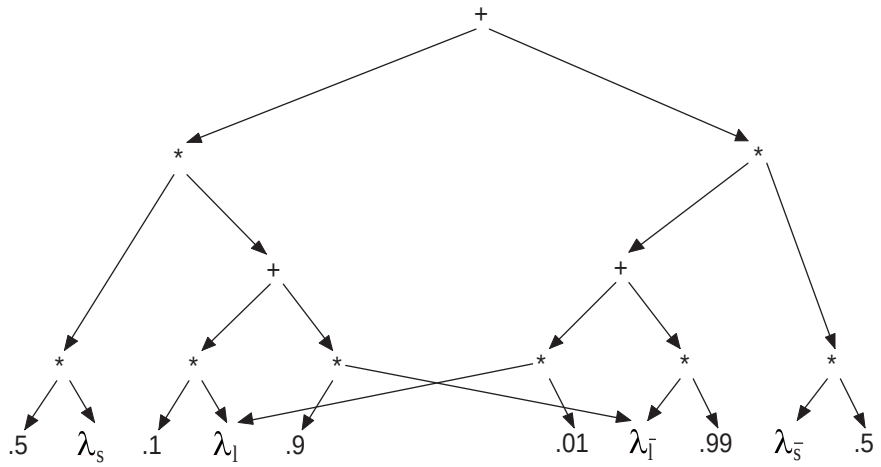


Figure 2.13: The polynomial of Equation 2.19, shown in graph form.

can be calculated from the derivative of its corresponding posterior probability:

$$P(x|e) = \frac{\partial F(\mathbf{e})/\partial \lambda_x}{F(\mathbf{e})} \tag{2.20}$$

Calculating $F(\mathbf{e})$ and $\partial F(\mathbf{e})/\partial \lambda_x$ from the AC requires a simple, rule-based algorithm (for details, see [42]). In fact, the partial derivative can be calculated for all values of all variables simultaneously. This means that once the algorithm completes, all posterior probabilities can be found in constant time.

Constructing an AC is very similar to the construction of a Q-DAG: modify an inference algorithm, replacing all arithmetic operations with node constructions. For any Bayesian network, an arithmetic circuit can be constructed such that its size is asymptotically the same as the complexity of JTP: $\mathbf{O}(n \exp(w))$ [16]. The time complexity of computing over the arithmetic circuit is linear on its size.

The primary advantage of AC in terms of calculating posterior probabilities is that the precompiled structure admits a very simple computation algorithm, like the Q-DAG algorithm, but with the added advantage of calculating all posterior probabilities simultaneously. However, like Query-DAGs, arithmetic circuits tend to be large, and as such are only applicable in environments where memory permits.

2.6 Summary

This chapter presented an indepth look at inference in Bayesian networks. Three classes of inference algorithms were considered. The first of these, the standard algorithms (elimination, junction-trees) are the most popular, probably due to their time efficiency. The second class, conditioning, is a linear-space approach to inference, at the cost of a time penalty for repeated calculation. Finally, precompiled inference structures were shown, which can further improve the time efficiency of standard algorithms, at the potential cost of more space.

One intent of this chapter, in addition to providing the reader with the necessary background, was to clearly show the resource tradeoffs in Bayesian network inference. These tradeoffs address some of the conflicts to using Bayesian networks that were addressed in the introduction. For example, a user in a highly space-constrained system might utilize some form of conditioning, while in a real-time system, the speed of the standard algorithms might be best suited, while an application running on a primitive architecture may call for the simplicity of precompiled structures.

The goal of the next four chapters is to reduce the severity of some of these tradeoffs. The data structure presented in the following chapter is a precompiled structure that utilizes conditioning to mitigate the space requirements of precompiled structures. Such a system combines the simplicity of precompiled structures with the space efficiency of recursive decomposition. The subsequent chapters are meant to mitigate the time penalties that we incur as a result of using conditioning. These include optimizations to the global structure (Chapter 4), context-specific optimization (Chapter 5), and caching (Chapter 6).

Chapter 3

Conditioning Graphs

This chapter presents *conditioning graphs*, which are compilations of a Bayesian network from which probabilities can be computed. Conditioning graphs enjoy several advantages over other methods of inference. As structures that use conditioning, they typically requires less space than JTP and VE, both for storage and for computation. As a recursive decomposition, they have a small memory footprint, and can employ time/space tradeoff techniques. However, conditioning graphs have their own unique advantages, even over other recursive decompositions. The conditioning graph, along with the inference algorithm, is a set of low-level instructions common to most computers, making them portable to virtually any machine. As well, there are no Bayesian network-specific elements present in this structure, which reduces the requirements for expertise to implement. The simplicity of the structure allows the inference operation to be easily and quickly measured in tangible quantities, such as measuring time by instructions, and size of the structure in bytes.

This chapter introduces the basic design of conditioning graphs. The following chapters will be dedicated to optimizing this initial design. In particular, Chapter 4 considers general optimizations to the system, Chapter 5 considers application-specific optimizations, and Chapter 6 examines caching methods for conditioning graphs.

3.1 Elimination Trees

Our notation remains the same as before, with some exceptions: from this point on, we assume that the domain of a random variable X is the set of integers from 0 to

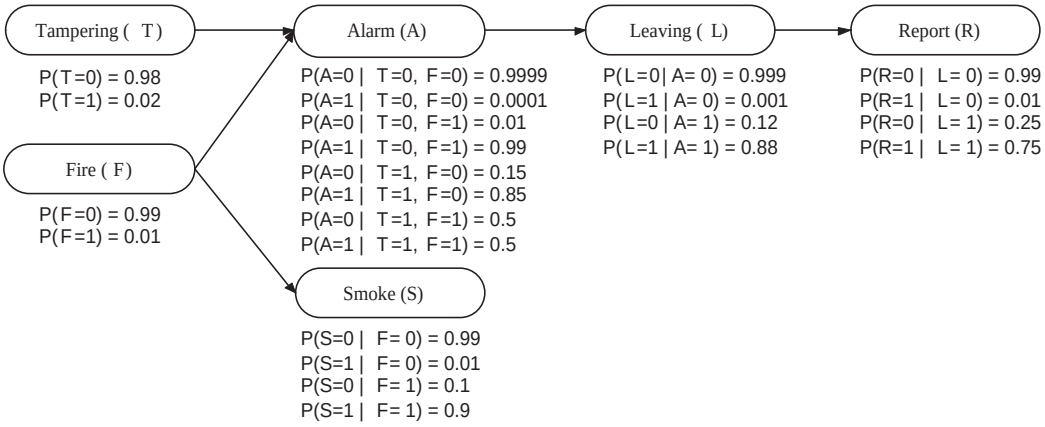
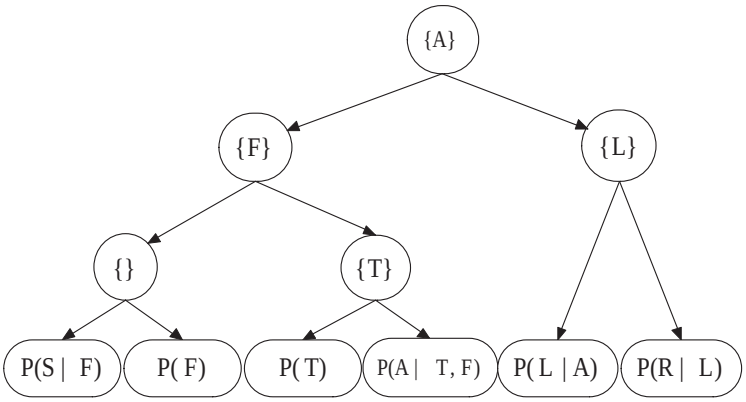


Figure 3.1: The *Fire* Bayesian network (taken from Poole et al. [53])

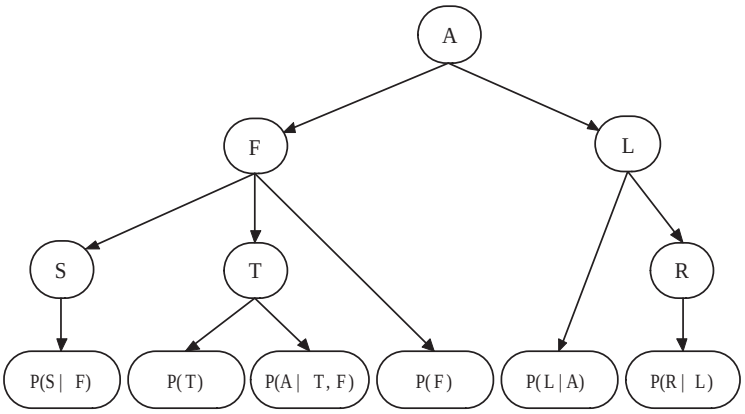
$m_X - 1$, where m_X is the domain size of X. Such an assumption is common, but requires a mapping from actual domain values to non-negative integers. Since all of our examples in this document use only boolean values, we assume that positive values (true, yes) are given the value 1, while negative values (false, no) are given the value 0. We will use the notation $x_i^{\mathbf{c}}$ to indicate the value of X_i in context $\mathbf{c}$. For example, if $\mathbf{c} = \{X_1 = 0, X_2 = 1\}$, then $x_1^{\mathbf{c}} = 0$ and $x_2^{\mathbf{c}} = 1$.

To simplify the presentation of our algorithms, we also change the notation for Bayesian networks. From here forward, we will represent a Bayesian network as a tuple $\langle \mathbf{X}, \mathbf{\Phi} \rangle$, where $\mathbf{\Phi} = \{\phi_1, ..., \phi_n\}$ is the set of CPTs, that is, each $\phi_i \in \mathbf{\Phi}$ is the CPT of X_i. This differs from our previous representation as the edges of the graph and the joint probability are implicitly represented in the set $\mathbf{\Phi}$. Figure 3.1 shows an example of a Bayesian network, and the CPTs associated with each variable, which we use as a running example throughout this chapter.

In the previous chapter, we introduced recursive decompositions. Recall that recursive decompositions partition a network by conditioning on a subset of its variables (such a subset of variables is called a *cutset*). Each of these components can be decomposed again, until each component in the base case is a single variable (with its associated distribution). Figure 3.2(a) shows an example of a recursive decomposition (a dtree [15]) for the *Fire* example. Note that we show only the cutset at each internal node.



(a) A recursive decomposition of the *Fire* network.



(b) An elimination tree for the *Fire* network.

Figure 3.2: Two decompositions of the *Fire* network.

An *elimination tree* is a tree whose leaves and internal nodes correspond to the CPTs and variables of a Bayesian network, respectively. The tree is structured such that all CPTs containing variable X_i in their domain are contained in the subtree of the node labeled with X_i. Each internal elimination treenode contains exactly one variable from the Bayesian network. Figure 3.2(b) shows a possible elimination tree for the *Fire* network.

The primary difference between an elimination tree and other recursive decompositions is that the variable at an elimination treenode does not necessarily represent a 'cutset.' Another difference is that the leaf variables of the Bayesian network are represented in the nodes of the elimination tree, whereas they are not in the cutsets of the dtree nodes (since a leaf node in a Bayesian network has no outgoing arcs, it can not belong to a cutset). The stated differences allow for simple low-level implementation, a primary goal of this project.

In the following discussion, we will use the notation for elimination trees and elimination treenodes interchangeably. That is, we can refer to T both as a tree and node. When it refers to a node, we are referring to the root of T.

One can construct an elimination tree directly from a variable ordering, using a variant of Variable Elimination. Figure 3.3 gives an algorithm for constructing an elimination tree from a Bayesian network $\langle \mathbf{X}, \mathbf{\Phi} \rangle$. Note that given an elimination tree T, $dom(T)$ is the union of the variables of the CPTs contained in T's leaves. An internal node represents the marginalization of its variable label, and the children of the node represent the distributions that would be multiplied together, were this standard variable elimination. One could also say that an internal node is labeled with a variable, but represents a distribution. This approach is also used by Darwiche when constructing dtrees [15] - although the labeling of nodes with cutsets is done as a post-construction step (after the entire tree has been assembled).

An example of the tree construction process is given in Figure 3.4. Initially, there is an elimination treenode for each CPT in the Bayesian network (Figure 3.4(a)). When a variable X is marginalized, a node is constructed with X as its label, and any partial elimination tree that contains a CPT whose domain includes X becomes

```
elimtree(⟨X, Φ⟩)
 1.  T ← {}
 2.  for each φ ∈ Φ do
 3.    Construct a leaf node T_φ containing φ
 4.    Add T_φ to T
 5.  for each X_i ∈ X do
 6.    Select the set T_i = {t ∈ T | X_i ∈ dom(t)}
 7.    Remove T_i from T
 8.    Construct a new internal node t_i whose children are T_i
 9.    Label t_i with X_i, and add it to T
10.  return T
```

Figure 3.3: Pseudocode for generating an elimination tree from a Bayesian network.

a child of the new node (Figures 3.4(b) - 3.4(g)).

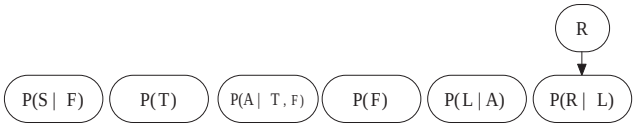
Notice that the algorithm in Figure 3.3 returns a set of trees, rather than a single tree. In the event that the network is not connected, the number of disconnected components will correspond to the number of trees returned by *elimtree*. For the following discussion, we consider the case where the elimination tree is a single tree. Cases where multiple trees occur are examined in Chapter 5.

To calculate probabilities from an elimination tree, we define algorithm $\mathcal{P}$ (see Figure 3.5). $\mathcal{P}$ takes as parameters a node from an elimination tree and a context, and returns a single value, which we will prove is the probability of that context. We use the following notation: if T is a leaf node, then ϕ_T represents the CPT at T. If T is an internal node, X_T represents the variable labeling T, and ch_T represents the children of T in the elimination tree.

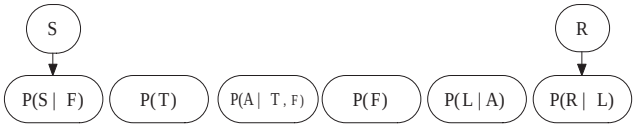
The following theorem relates the probabilities of interest and the algorithm $\mathcal{P}$. A proof of the theorem can be found in Appendix A.

Theorem 3.1.1. *Given a Bayesian network* $\langle \mathbf{X}, \mathbf{\Phi} \rangle$ *and an associated elimination*

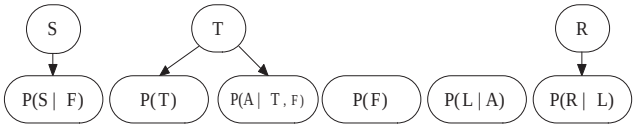
(a) Initialization. Each CPT is represented in an elimination treenode.



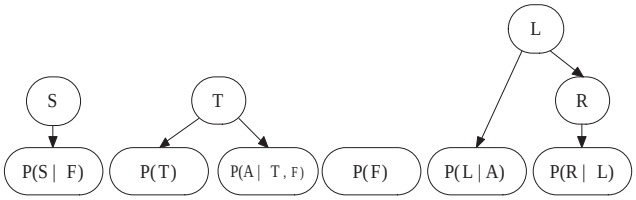
(b) After variable R is chosen.



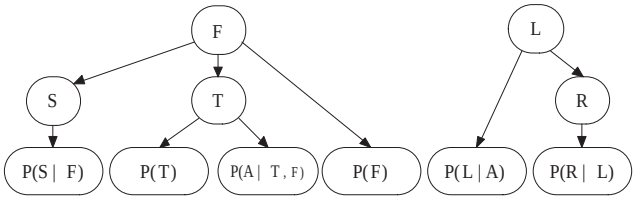
(c) After variable S is chosen.



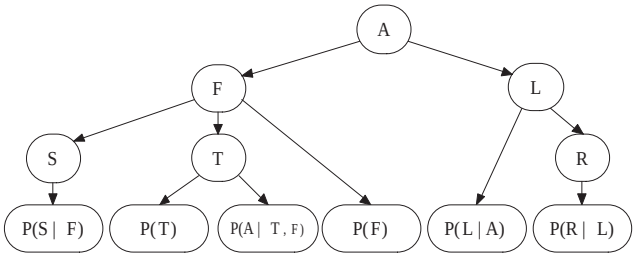
(d) After variable T is chosen.



(e) After variable L is chosen.



(f) After variable F is chosen.



(g) After variable A is chosen.

Figure 3.4: Elimination tree construction using the *elimtree* algorithm, with the elimination ordering $[R, S, T, L, F, A]$.

```
P(T, c)
 1.  if T is a leaf node
 2.    return φ_T(c)
 3.  elseif X_T is instantiated in c
 4.    Total ← 1
 5.    for each T' ∈ ch_T while Total > 0
 6.      Total ← Total * P(T', c)
 7.    return Total
 8.  else
 9.    Total ← 0
10.    for each v_T ∈ D(V_T)
11.      Total ← Total + P(T, c ∧ {v_T})
12.    return Total
```

Figure 3.5: Code for processing an elimination tree given a context.

tree T:

$$P(x_q|\mathbf{c}) = \alpha\mathcal{P}(T, \{x_q\} \wedge \mathbf{c}) \tag{3.1}$$

where $\alpha = P(\mathbf{c})^{-1}$ is a normalization constant.

The major advantage of recursive decompositions (and conditioning in general) is their efficient use of space. We summarize this in the following theorem.

Theorem 3.1.2. *Given a Bayesian network and a corresponding elimination tree T, $\mathcal{P}(T, \mathbf{C} = \mathbf{c})$ makes $\boldsymbol{O}(n \exp(d))$ recursive calls and requires $\boldsymbol{O}(n \exp(f))$ space, where d is the height of the elimination tree, and f is the size of the largest family in the Bayesian network.*

Proof. In order to store the CPTs of the Bayesian network, we require $\mathbf{O}(n \exp(f))$ space. Hence, it suffices to show that the storage required to store and compute over the elimination tree does not exceed this bound. Because the extra storage for the elimination tree is linear in the number of nodes on the network, the storage bound holds.

For time complexity, each internal node in the elimination graph corresponding to an unobserved variable of size m recursively calls $\mathcal{P}$ on its children m times - once for each instantiation of its variable. This means that a node with d nodes in its ancestry gets called $\mathbf{O}(\exp(d))$ times. □

Theorem 3.1.2 demonstrates the relationship between the depth of the tree and the complexity of the algorithm $\mathcal{P}$. The depth of the tree is a consequence of the order in which the variables are selected during the *elimtree* algorithm. We address the problem of finding good orderings for optimizing elimination tree height in Chapter 4.

3.2 Conditioning Graphs

In this section, we will give a low-level representation for a Bayesian network as an elimination tree, and a compact efficient implementation of the algorithm $\mathcal{P}$.

A CPT is a function that maps a context to a probability. In this work, a table-based representation of each CPT is used, as it facilitates an easy look-up scheme. There exist several different CPT representations throughout the literature, that offer advantages such as the ability to represent and exploit context-specific independence (CSI) [9, 52, 54]. Exploiting CSI in conditioning graphs is a future project.

In a linear representation of a CPT, the entries are typically sorted according to some variable ordering, where the entries are sorted on the most significant variable, then the second-most significant variable, and so on. Let ϕ be a CPT containing variables $\mathbf{C} = \{X_1, ..., X_k\}$. Let $\rho = [X_{(1)}, ..., X_{(k)}]$ represent an ordering of those variables. We can use ρ to define an ordering to the contexts of $\mathbf{C}$ using the following rule: given two contexts $\mathbf{c}, \mathbf{d}$ over the variables in $\mathbf{C}$, $\mathbf{c} < \mathbf{d} \Leftrightarrow \exists j$ s.t. $(x^{\mathbf{c}}_{(j)} < x^{\mathbf{d}}_{(j)}) \wedge \forall i < j (x^{\mathbf{c}}_{(i)} = x^{\mathbf{d}}_{(i)})$. Figure 3.6 shows the *Alarm* CPT from our *Fire* example, stored according to different variable orderings.

If the entries of a CPT are stored in a zero-based array (with no empty spaces) and sorted according to $\rho = [X_{(1)}, ..., X_{(k)}]$, then we can calculate the index of a

T	F	A	P(A | T, F)
0	0	0	0.9999
0	0	1	0.0001
0	1	0	0.01
0	1	1	0.99
1	0	0	0.15
1	0	1	0.85
1	1	0	0.5
1	1	1	0.5

(a) ρ = (T, F, A).

A	F	T	P(A | T, F)
0	0	0	0.9999
0	0	1	0.15
0	1	0	0.01
0	1	1	0.5
1	0	0	0.0001
1	0	1	0.85
1	1	0	0.99
1	1	1	0.5

(b) ρ = (A, F, T).

T	A	F	P(A | T, F)
0	0	0	0.9999
0	0	1	0.01
0	1	0	0.0001
0	1	1	0.5
1	0	0	0.15
1	0	1	0.5
1	1	0	0.85
1	1	1	0.5

(c) ρ = (T, A, F).

Figure 3.6: The *Alarm* CPT sorted according to different variable orderings.

context $\mathbf{c} = [x_{(1)}, ..., x_{(k)}]$ as:

$$index([x_{(1)}, ..., x_{(i)}]) = x_{(i)} + m_{(i)} * (index([x_{(1)}, ..., x_{(i-1)}])) \tag{3.2}$$

where $index([]) = 0$ is the base case, and $m_{(i)}$ is the size of $X_{(i)}$'s domain. Storing in this order eliminates the need for storing contexts along with their probabilities; we only need to know in advance the variable ordering. If we choose an ordering that is consistent with the path from root to leaf in the elimination tree, then as we traverse the tree, we can update the current index of each CPT, using Equation 3.2. More specifically, when a node N with labeling variable X_i is visited, then for each CPT that includes X_i in its definition, we multiply the current index of that CPT by m_i, and add to it the current value of X_i. This requires associations between variables and distributions, which we represent as a second set of arcs at each internal node, referred to as *secondary pointers* (call the original pointers *primary pointers*). The secondary arcs are added according to the following rule: *there is an arc from an internal node A to leaf node B iff the variable X associated with A is contained in the domain of the CPT associated with B*. The number of secondary arcs emitting from a node with variable X is equivalent to $|ch_X| + 1$, where ch_X refers to the number of arcs emitting from X in the Bayesian network. Cumulatively, the number of secondary arcs in the entire structure is $e + n$, where n and e are the number of nodes and arcs in the Bayesian network, respectively.

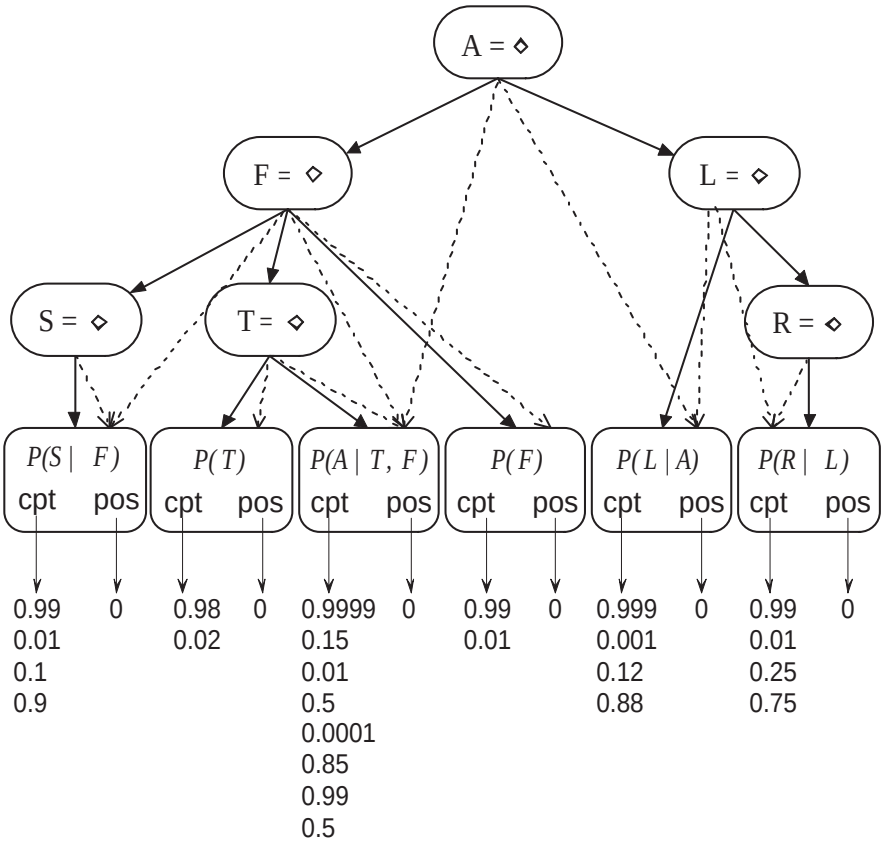


Figure 3.7: The conditioning graph.

An example of the final structure is shown in Figure 3.7, which we refer to as a *conditioning graph*. Note that at each leaf, we store the CPT as an array of values, and the index as an integer variable, which we call *pos*. In each internal node, we store a set of primary pointers (from the elimination tree), a set of secondary pointers, and an integer representing the current value of the node's variable.

We maintain one global context over all variables, denoted as $\mathbf{g}$. Each variable X_i is instantiated in $\mathbf{g}$ to a member of $\{0, \ldots, m_i - 1\} \cup \{\diamond\}$. The symbol $\diamond$ (borrowed from Darwiche and Provan [18]) is a special symbol that means the variable is unobserved. Initially, all nodes are assigned $\diamond$ in $\mathbf{g}$, as no variables have been instantiated. To calculate $P(E_1 = e_1, \ldots, E_k = e_k)$, we set $E_i = e_i$ in $\mathbf{g}$ for $i = 1$ *to* k. Any node whose variable does not have a value at the start of the algorithm will be "conditioned", meaning that we process the node once for each value of the variable.

Figure 3.8 shows *Query*, a more detailed implementation of $\mathcal{P}$ that computes over conditioning graphs. Note that we use dot notation to refer to the members of the variables. For a leaf node N, we use $N.cpt$ and $N.pos$ to refer to the CPT and its current index, respectively. For an internal node N, we use $N.primary$, $N.secondary$, $N.size$, and $N.value$ to refer to the node's primary children, secondary children, variable domain size, and variable value, respectively. The member *value* at each internal node can also represent the input from the programmer. To set evidence $X = x_i$, the programmer would have to set $N.value$ to the appropriate value for the node N labeled with variable X before calling *Query*.

When the value of a variable V changes, either through observation or conditioning, the value at V's node needs to be set appropriately. Figure 3.9 gives a function, *SetEvidence*, that performs this task for the user. Note that the function is only one line of code, thus, encapsulating this line of code in its own function may seem inefficient. However, it serves two purposes:

1. It provides a useful abstraction for users not interested in the details of the implementation.

2. As we improve on the conditioning graph library over the course of this docu-

```
Query(N)
 1.  if N is a leaf node
 2.    return N.cpt[N.pos]
 3.  else if N.value ≠ ⋄
 4.    for each S′ ∈ N.secondary
 5.      S′.pos ← S′.pos ∗ N.size + N.value
 6.    Total ← 1
 7.    for each P′ ∈ N.primary while Total > 0
 8.      Total ← Total ∗ Query(P′)
 9.    for each S′ ∈ N.secondary
10.      S′.pos ← S′.pos/N.size
11.    return Total
12.  else
13.    Total ← 0
14.    for N.value ← 0 to N.size − 1
15.      Total ← Total + Query(N)
16.    N.value ← ⋄
17.    return Total
```

Figure 3.8: The *Query* algorithm, which takes the root of the conditioning graph, and recursively computes the probability of the current context. Note that on Line 10, we are using integer division, so the fractional part of the result is dropped.

ment, the *SetEvidence* function will become more elaborate, making a function representation appropriate. By initially providing such a function, we maintain continuity throughout this document.

```
SetEvidence(N, i)
 1.   N.value ← i
```

Figure 3.9: The *SetEvidence* algorithm, which takes a node N containing variable V, and sets V's value to i, where $i \in \{0, ..., m_V - 1\} \cup \{\diamond\}$.

3.3 Implementation Details

3.3.1 Compilation

The previous section introduced conditioning graphs. Conditioning graphs can be seen as a compilation of inference in Bayesian networks - its structure represents a schedule for the elimination of variables. So far, we have introduced what conditioning graphs are. The section following this one illustrates how conditioning graphs meet the goals listed in Chapter 1. However, it may not be clear how the conditioning graph is initially generated. That is the purpose of this section.

The construction of conditioning graphs is handled by a compiler. The compiler input is a Bayesian network. It first constructs an elimination tree (using the algorithm in Figure 3.3). It then adds secondary links, and creates the integers and arrays necessary for computation. The output is a conditioning graph, and the corresponding code for computing over that algorithm.

The compilation of the conditioning graph can be achieved automatically, and therefore is transparent to the programmer using the conditioning graph. This is important, as one of the goals of conditioning graphs is to abstract away the details of Bayesian network computation from the user. The compilation step of conditioning graphs accomplishes this step: computation and decisions that are strictly inference related are taken care of during compilation. This not only accomplishes

the desired abstraction, but allows the programmer to accurately assess the runtime and memory requirements of the algorithm: there is no ambiguity in the schedule of computation. This assumes that the compiler incorporates expertise for inference in Bayesian networks, but since it is a compile time step, the complexity of the compiler is of little consequence to the end user of conditioning graphs.

For the remainder of the document, any structural extensions to the conditioning graph will be assumed to take place at compile time. As an example, the secondary indices of Section 4.3 are assumed to be computed at compile time. The structural extensions are optional, and we can augment the inputs of the compiler with flags, to specify with of the these augmentations we would like to include.

3.3.2 Implementation

To examine conditioning graphs and compare them to other methods, we developed several implementations. As a demonstration of the portability of conditioning graphs, the first implementation is given in a high-level language. We chose the C programming language because it is closer to a machine-level language than most other high-level languages. Appendix B gives the implementation details, which includes a translation of the functions *Query* and *SetEvidence*, and an example application.

As the implementation demonstrates, the translation to a high-level programming language can be done line by line (the line numbers corresponding to the pseudocode are given in comments), making it simple enough for any coder, regardless of Bayesian-network background. The limited number of lines in the code (17 for the *Query* function) also make this translation simple (translating a large program line by line would be tedious and error-prone). Note that no libraries have been imported, and only basic coding constructs have been used. Hence, not only does a coder need not be an expert in Bayesian networks to implement the functions, but they do not need to be experts in the implementation language either; a basic working knowledge is sufficient.

As a demonstration of the small memory requirements of the algorithm, we looked

at the size of a compiled implementation. We chose the MIPS architecture for our example compilation. MIPS is a machine language for a RISC architecture, with 32 machine registers and a FPU. The operations are typical of other machine languages, which means the size of the MIPS implementation will be a good indication of the actual size requirements of the compiled functions. Appendix C gives the MIPS implementation of the conditioning graph algorithm, including the example from the end of Appendix B. In this particular implementation, the *Query* function compiles to 76 MIPS instructions, while *SetEvidence* compiles to just 2. Assuming each instruction is one word, the inference code requires $78 \times 4 = 312$ bytes; by comparison, the Netica library DLL requires 811 KB of storage on a Windows platform (as a DLL), and almost 2 MB on a Unix platform. [1]

In the following discussion, when discussing the merits of the conditioning graph architecture, we will refer back to these implementations. Note that these are just two examples of a conditioning graph implementation, and many others are possible.

3.4 Discussion

The conditioning graph architecture solves many of the issues addressed in the introduction. We consider each issue in turn.

Space Complexity. As mentioned, the most popular methods of inference in Bayesian networks are junction-tree processing and variable elimination. Both methods require $\mathbf{O}(n \exp(w))$ memory space, where w is the induced width of the variable elimination ordering. Such memory requirements reduce the portability of the application to space-conservative applications, such as embedded systems.

Conditioning graphs, on the other hand, require $\mathbf{O}(n \exp(f))$ memory, where f is the size of the largest family. Since $f \leq w$, conditioning graphs will never re-

[1] Note that this comparison between commercial libraries such as Netica and conditioning graphs compares the program sizes, not functionality. It should be acknowledged that the large commercial libraries contain many additional functions for computing in Bayesian networks, not just posterior probability computation. Hence, if this extra functionality is desired, then choosing one of these libraries is appropriate (assuming available memory space).

quire more space than the other methods; in practice, f is typically smaller than w, which means an exponential reduction in the memory requirements. Table 3.1 compares the space requirements of junction-tree propagation, variable elimination, and conditioning graphs over some Bayesian networks commonly used for testing and comparison [1]. The memory shown assumes 4 bytes of memory for all data types, including pointers, probabilities, and integers. To measure the size requirements of junction tree propagation, the networks were compiled on the Netica software package, a leading commercial application for computing over Bayesian networks using junction-tree propagation [47]. For our implementation of variable elimination, we store the original CPTs, as well as an intermediate distribution for each eliminated variable. The elimination ordering for each network was the same as the one used by Netica for compilation. There are two notes to make about the comparison. First, the space requirements listed for JTP and VE accounted only for the sizes of the distributions stored by the two algorithms. No account was made for other details of the algorithms, such as context representation, CPT representation (indexing data types, stacks and queues for message scheduling, etc). By contrast, the space requirements listed for conditioning graphs encompass its entire memory requirements, including all pointers, indexing integers, integers representing context, and so on. Second, the reported space requirements of JTP and VE do not consider any space optimizations, which means in some implementations, these numbers could be reduced. However, they are still bounded from below by $\mathbf{O}(n \exp(w))$.

As the table shows, the amount of storage required for the conditioning graph is a small fraction of the storage required by the standard approaches. All but two networks can be stored with a conditioning graph in under one megabyte. In all but one case (Mildew), the conditioning graph storage represented less than 5% of the JTP/VE storage requirements.

The second column of the table shows the total size of the CPTs in each Bayesian network. We can also see from the data that the storage requirements for conditioning graphs is only slightly larger than for the actual Bayesian network. Hence, the conditioning graph model is only slightly larger than the actual Bayesian network

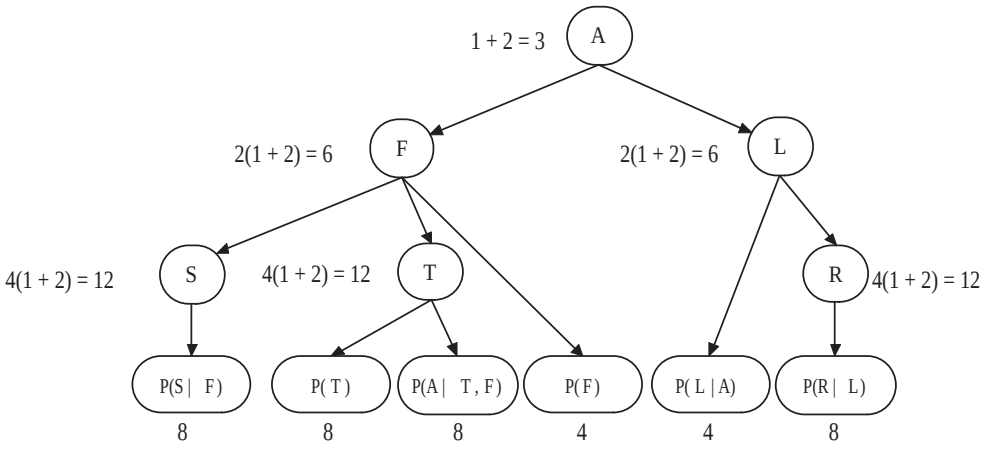
Table 3.1: Size requirements (in MB) of JTP, VE, and Conditioning Graph (CG) storage and computation.

Network	Size	JTP	VE	CG
Barley	0.4966	107.1	99.85	0.4978
Diabetes	1.7580	42.82	44.13	1.769
Link	0.0782	4333	5770	0.0944
Mildew	2.0872	50.96	16.13	2.088
Munin1	0.0743	835.8	825.5	0.0786
Munin2	0.3201	18.16	17.75	0.3406
Munin3	0.3275	14.46	14.58	0.3489
Munin4	0.3745	69.89	74.55	0.3962
Pigs	0.0321	3.081	3.189	0.0415
Water	0.0514	32.82	33.49	0.0524

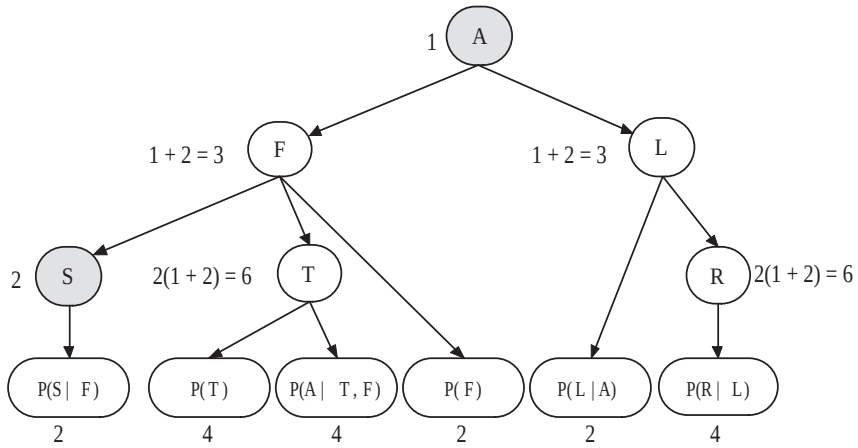
itself. Therefore, if an application has enough room to store the Bayesian network, then it is likely that it can accommodate a conditioning graph representation as well. As the table shows, this is not the case for JTP and VE, which in some cases require orders of magnitude more space for storage than the actual network does.

Portability. The conditioning graph algorithm uses simple instructions (common to most languages), and its entire specification is seventeen lines of code. As demonstrated by the implementation in Appendix B, the translation from the conditioning graph algorithm to a high-level language is very straightforward, and requires no background knowledge about Bayesian networks. Our example compiled version of the conditioning graph algorithm (Appendix C) is a fraction of a kilobyte, making it accessible to memory-limited environments.

Assessibility. In the conditioning graph algorithm, the computation is essentially governed by the elimination tree. Trees are familiar structures to programmers, and the elimination tree makes the amount of computation easy to assess. The number of times a node N will be called by the *Query* function is an easy recursive calculation. Let $C(N)$ be the product of the domain sizes of the unobserved variables in N's ancestry. If N's variable is observed, then the number of recursive calls to N is



(a) No evidence.



(b) *Alarm* and *Smoke* observed.

Figure 3.10: The *Fire* elimination tree. Number of recursive calls to each node is shown beside (or below) the node.

$C(N)$, otherwise, the number of recursive calls is $C(N) * (N.size + 1)$. Figure 3.10 shows the number of recursive calls for each node under two different contexts. Such a simple counting scheme makes it easy for even a non-expert to ascertain approximate runtimes, and given the simplicity of the *Query* algorithm implementation (Appendix B and C), the runtime of the algorithm can be assessed from the number of recursive calls quickly and accurately.

A similar argument applies to the space requirements of the algorithm. The number of nodes and arcs in the conditioning graph give a very accurate account of the extra space required by the algorithm (in addition to storing CPTs). This

number is obtainable directly from the structure, without requiring any expertise in Bayesian network inference.

Time. Compiled structures, such as junction trees and dtrees, perform much of their computation while the structure is being created. For example, in junction-tree implementations, the distributions at each cluster can be multiplied at construction time. Dtrees have an advantage over query-based elimination algorithms in that their structure maintains relationships between the distributions, hence reducing the need to search for distributions with particular variables before marginalization. Furthermore, Monti and Cooper [44], as a compilation step, fill the caches of their dtree nodes by performing inference over no evidence, which substantially reduces inference time by precalculating certain values. Conditioning graphs are a type of recursive decomposition, and therefore inherit these same advantages (we will examine how to cache with conditioning graphs in Chapter 6).

However, while conditioning algorithms typically use less space than JTP/VE, they also require more runtime. Conditioning graphs are no exception, and this penalty can mean orders of magnitude degradation in performance when compared to the standard algorithms. The remainder of this document focuses on optimizing the conditioning graph model, in order to make it more competitive with the more popular algorithms, while still maintaining the properties set out in this chapter.

3.5 Summary

This chapter presented conditioning graphs, a low-level representation of inference in Bayesian networks. The conditioning graph structure requires only slightly more memory to store it than the original Bayesian network. The algorithm for computing probabilities from a Bayesian network, because of its recursive character, has a very small memory footprint, allowing it to fit into even the most modest architectures, such as embedded systems (e.g., digital cameras), video games or multi-agent systems. The low-level nature of the algorithm, combined with its compact size, makes

it universally implementable, on any architecture and by any programmer. These properties allow the conditioning graph model to meet most of the requirements set forth in the first chapter.

Conditioning graphs abstract the details of inference in Bayesian networks from the user, without abstracting away the implementation details. This accessibility of the code, as well as contributing to its portability, allows the user to assess the program's time and space requirements exactly. While these details are present for the interested user, the uninterested user can still choose to ignore them; and work only with the interface, which is simply a pair of functions, *SetEvidence* and *Query*.

As mentioned, the remainder of the content of this document focuses on optimizing the conditioning graph model, in order to make it more competitive with elimination algorithms, in terms of runtime. Specifically, Chapter 4 deals with general optimizations, which apply to the model in general. Chapter 5 deals with application-specific optimizations, and shows how we can exploit some well known independencies of the Bayesian network given specific contexts and queries.

Chapter 4

General Optimizations

4.1 Introduction

In the last chapter, we presented *conditioning graphs*. A conditioning graph is a compilation of a Bayesian network used for computing posterior probabilities. Conditioning graphs allow us to compute probabilities from a Bayesian network without requiring monolithic runtime libraries, or the implementation of complex inference techniques such as VE or JTP.

Despite their advantages, inference algorithms for Bayesian networks based on conditioning typically require more computation time than VE and JTP. This has at least two implications for the conditioning graph model:

1. While conditioning graphs may provide Bayesian network inference to architectures of limited memory (e.g., embedded devices), the time required to compute probabilities from the model may simply be impractical.

2. There is a strong argument for the value of time over memory (especially as the price of memory continues to fall). By this argument, it would always be beneficial to choose JTP and VE for inference wherever possible, ignoring the overhead and memory requirements.

It should be noted that recursive decomposition algorithms such as conditioning graphs can never be asymptotically "faster" than standard algorithms such as JTP or VE. More specifically, JTP and VE are asymptotically equivalent to recursive decomposition algorithms that employ caching. Our goal therefore is to improve the conditioning graph model to "close the gap" between the runtimes of the more

popular inference algorithms and conditioning graphs as much as possible, while still maintaining the same space complexity.

This chapter deals with general improvements to the conditioning graph model. These improvements do not depend on application-specific details (the topic of Chapter 5). We consider three optimizations. The first optimization examines methods for building shallower elimination trees. The second is an indexing improvement that produces a considerable speedup in inference while introducing a small constant to the space complexity. The final optimization is an extension to the algorithm that ignores leaf variables of the Bayesian network when they are unobserved, which also provides considerable runtime speedup at the cost of another small constant.

4.2 Building Shallow Elimination Trees

The time complexity of inference using a conditioning graph is exponential on the height of its underlying elimination tree. Hence, reducing the height of elimination trees results in an exponential speedup of the algorithm. In this chapter, we examine methods for reducing the heights of elimination trees. We first look at methods for balancing dtrees [15, 17], and demonstrate a simple transformation between dtrees and elimination trees, such that the time complexity of inference in the elimination trees after conversion is the same as in the dtree. We also demonstrate two new heuristics for constructing recursive decompositions, based on greedy search, and show empirically that these are better than the method suggested by Darwiche and Hopkins [17] for tested networks when no caching is employed.[1]

4.2.1 Dtrees to Elimination Trees

As discussed in Section 2.4, a *dtree* is a recursive decomposition of a Bayesian network, where each internal node has exactly two children. The number of variables at each node is not restricted to one variable as it is in elimination tree nodes. Figure

[1]These results were presented in Grant and Horsch [31].

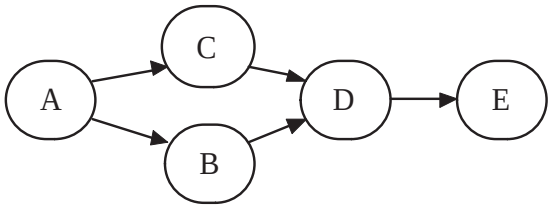


Figure 4.1: An example Bayesian network.

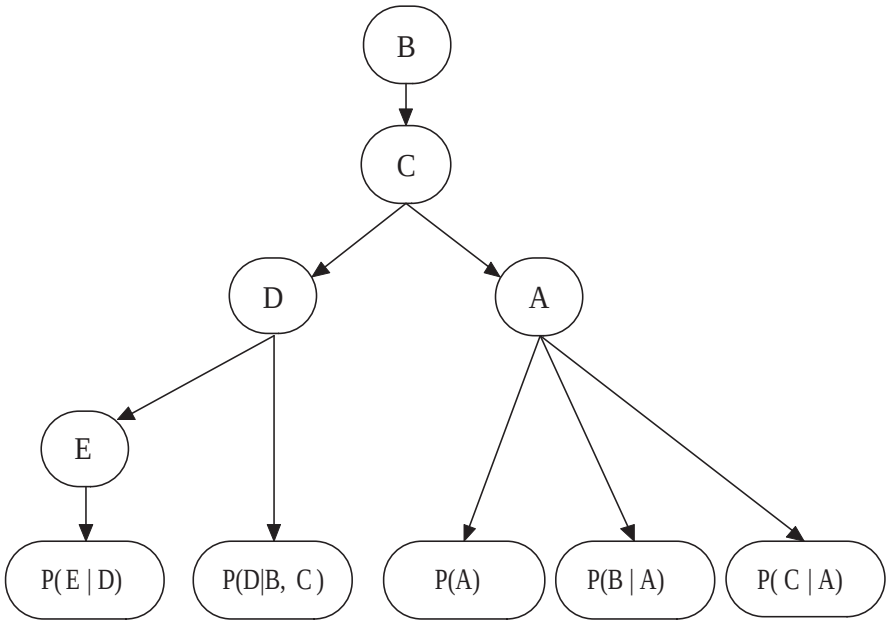


Figure 4.2: An elimination tree for the Bayesian network in Figure 4.1

4.1 shows an example Bayesian network, while Figures 4.2 and 4.3 show a possible elimination tree and dtree for that network, respectively.

The time complexity of computing probabilities in a dtree when no caching is used is $\mathbf{O}(n\ \exp(wd))$ [15], where n is the number of Bayesian network variables, w is the size of the largest cutset, and d is the maximum depth of the tree. If the tree is balanced, then $d = \mathbf{O}(\log n)$. There are two algorithms for balancing dtrees. The first [15] involves constructing a dtree using variable elimination (in the same manner as elimination trees are constructed), and then balancing it using contraction [43]. The second involves directly computing a balanced tree using hypergraph partitioning [17].

The similarity between dtrees and elimination trees suggests that a well-balanced

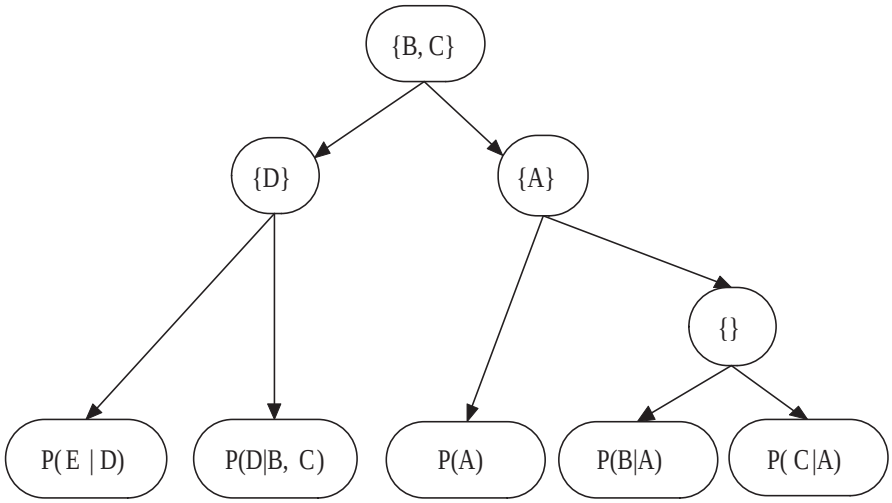


Figure 4.3: A dtree for the Bayesian network in Figure 4.1

dtree might lead to a well-balanced elimination tree. Transforming a dtree to an elimination tree is straightforward. We give a transformation method, and then show that the complexity of the resulting elimination tree is the same as the original dtree.

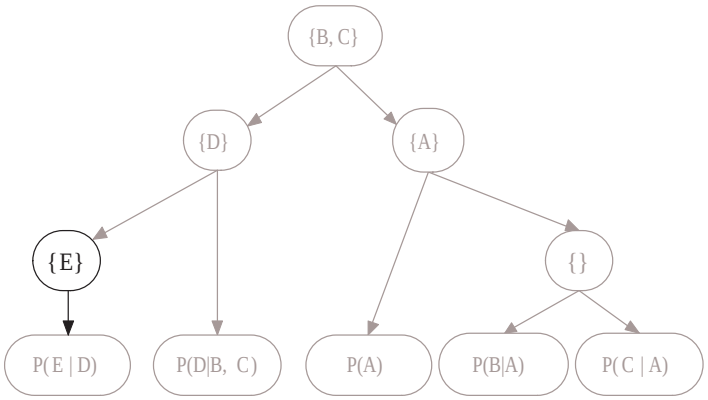
The conversion from a dtree to an elimination tree requires three steps.

1. For each leaf variable in the original Bayesian network, create a new node containing that variable, and insert it on the path directly above its CPT node. Figure 4.4(a) shows an example.

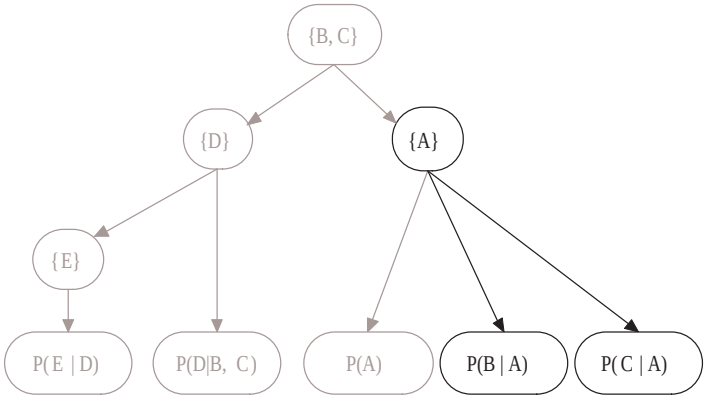
2. If a node N has no variables in its cutset, then N's children are promoted to be children of N's parent. Figure 4.4(b) shows an example.

3. If a node N has k variables in its cutset, where $k > 1$, then replace node N with a directed path of k nodes, each containing one variable from N's cutset. Figure 4.4(c) shows an example.

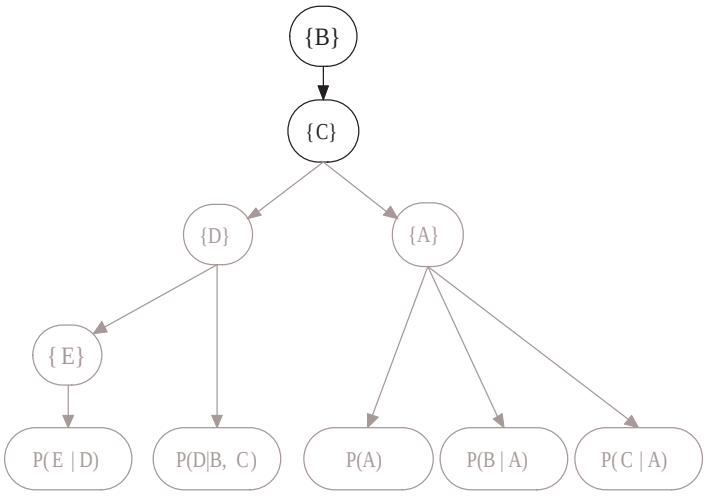
The order in which these steps are performed does not matter. After conversion, each cutset will contain exactly one variable, which is the variable that will label its corresponding node in the elimination tree (Figure 4.2).



(a) Step 1 in the conversion process. Note that a node containing the variable E has been inserted above E's cpt.



(b) Step 2 in the conversion process. Note that the empty node has been eliminated, and its children nodes are now children of A.



(c) Step 3 in the conversion process. Note that the node containing B and C has been replaced with a chain of nodes.

Figure 4.4: The dtree to elimination tree conversion process.

Lemma 4.2.1. *After converting a dtree of a Bayesian network using the above algorithm, the resulting structure is an elimination tree over the Bayesian network.*

Proof. Steps 2 and 3 in the conversion process ensure that each node in the elimination tree has only one variable. We need now show that the domain of each CPT is contained in the ancestry of that CPT's node in the elimination tree. By the property of dtrees, we know that this is true in the dtree for all variables that are not leaves in the Bayesian network. Since none of the steps change the partial ordering of the variables in the dtree, this will also be true in the elimination tree for those variables. Step 1 ensures this property for the leaf variables of the Bayesian network, which are not contained in the cutsets of a dtree. □

We now show that the time complexity of inference using the dtree and the corresponding elimination tree is the same.

Lemma 4.2.2. *After converting a dtree of depth d and cutset size w, the resulting elimination tree has a height of d_2, where $d_2 \leq wd + 1$.*

Proof. The theorem follows directly from the transformation. In Step 1, adding a leaf variable above a leaf node increases any path in the tree by at most 1. In Step 2, promoting nodes to a new parent does not increase the height. In Step 3, creating a chain out of a set of cutset variables increases the length of a path by at most $w - 1$, since the cutset size is bounded by w. Hence, since the longest path is bounded by d, and it can be increased by at most $d(w - 1) + 1$, the maximum length of a path in the elimination tree is $wd + 1$. □

Theorem 4.2.1. *The time complexity to compute posterior probabilities using a dtree is the same as the time complexity using an elimination tree constructed from that dtree.*

Proof. As mentioned, the time complexity of inference using a dtree of height d and width w is $\mathbf{O}(n\ exp(wd))$, where n is the number of variables in the Bayesian network. From the lemma, the elimination tree constructed from such a dtree is of height $wd+1$. Since the time complexity of inference over an elimination tree is exponential on its height, it follows that the two structures have the same complexity. □

Elimination trees can be built from dtrees, as above. However, this is not necessarily the best method for producing shallow elimination trees. In the next section, we propose an alternative procedure, and compare these methods in Section 4.2.3.

In closing, we note that the transformation algorithm from a dtree to an elimination tree can be reversed, so that a dtree can be produced from an elimination tree, and the structures have the same complexity for inference. This is important, since the complexity of computing over a dtree is the product of the height of the tree and the width of the cutsets. In Darwiche et al. [17], the authors explicitly compare the construction algorithms by each term, but not by product of these two factors. In contrast, the complexity of computing over elimination trees is a function only of height. Therefore, by minimizing the complexity of an elimination tree, we are minimizing the product of height and width in a dtree. Therefore any method developed to build good elimination trees can be used to build good dtrees. This is especially important if the dtree will be used without any caching of intermediate results.

4.2.2 Better Elimination Orderings

In Chapter 2, we discussed how inference algorithms such as JTP or VE require a good elimination ordering to obtain small cliques, or small intermediate distributions. Finding an optimal elimination ordering is NP-hard; heuristic methods, which are relatively fast, have been shown to give good results in most cases. Starting from a moralized graph, the *min-fill* heuristic chooses to eliminate the variable which would require the fewest edges to be added to the network during triangulation; the *min-size* heuristic chooses to eliminate the variable with the fewest number of neighbouring variables [34, 38].

These heuristics are not necessarily well suited for building recursive decompositions, especially when no caching will be performed during inference [15]. The heuristics try to minimize clique size, which is not directly related to the time complexity of inference over a decomposition structure such as an elimination tree. Consider the Bayesian network shown in Figure 4.5. Using the *min-fill* heuristic, we will always

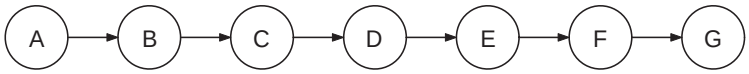


Figure 4.5: For this Bayesian network, an elimination ordering that is optimal for inference based on junction trees is the worst case for methods based on decomposition structures.

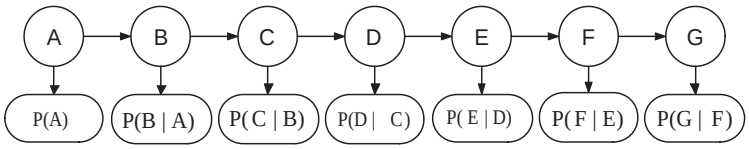


Figure 4.6: A worst case elimination tree for the Bayesian network in Figure 4.5, constructed using the min-fill heuristic.

remove a node from the end of the chain, which leads to the possibility of an elimination ordering such as G, F, E, D, C, B, A. This ordering is optimal for inference methods based on junction trees or variable elimination. However, it is the worst case for inference over elimination trees and dtrees. Figure 4.6 shows the elimination tree generated from this elimination ordering (the corresponding dtree is exactly the same, minus the node containing the leaf variable). The height of the elimination tree corresponds to the number of nodes in the network, making the complexity of inference $\mathbf{O}(n\ \exp(n))$.[2]

Darwiche shows that an unbalanced dtree can be balanced using rake and compress methods [43]. However, we take a more direct approach, trying to measure (and minimize) the height of the elimination tree as we construct it. Recall that an elimination tree is constructed iteratively (Figure 3.3), and when a variable is chosen as the root of a new elimination tree, all partial elimination trees that include this variable in their definition are made children of the chosen variable. We wish to choose the variables in such an order (Line 05) that the eventual elimination tree height is minimized.

The eventual height of the final elimination tree is estimated by taking the height of the current partial elimination tree being constructed (as a result of choosing a

[2]The best possible ordering chosen by min-fill for a chain of variables leads to an elimination tree of height $n/2$, which is still linear in the number of variables.

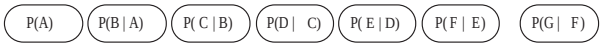
variable), and estimating the additional height above this tree as a result of later iterations. This is very similar to the way heuristics are used in A* search: the best choice minimizes f, which is the sum of current cost g with estimated remaining cost h. We define $g(T)$ as the current height of a given elimination tree T. The estimate $h(T)$ is the number of variables in the domain of T that have not yet been eliminated. This value corresponds exactly to the *min-size* heuristic of classical elimination order generation, and provides a lower bound on the remaining height of the tree. Therefore, when choosing a variable, we choose the variable that creates a partial elimination tree with the lowest cost (lowest sum of $g(T) + h(T)$), breaking ties with the current height of the tree $g(T)$.

We demonstrate this with an example (Figure 4.7). Initially, there is one partial elimination tree for each CPT (Figure 4.7(a)). The next variable to be added to the final elimination tree is the one whose partial elimination tree T minimizes $g(T) + h(T)$. At this point, the elimination tree of any variable will have height 1, therefore, $g(T) = 1$ for all variables. However, if A or G is selected, then the resulting elimination tree will only contain one non-eliminated variable, versus two for the other variables. Therefore, we eliminate one of these variables first (A in our example, Figure 4.7(b)).

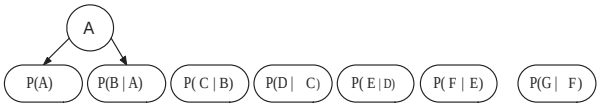
Figure 4.7(c) - 4.7(h) shows the remainder of the construction process using this heuristic. The tree produced shows a substantial improvement over simply using the *min-fill* or *min-size* heuristic. Although the tree produced in this example is optimal, the heuristic is not guaranteed to produce optimal trees in every case. We evaluate the performance of the heuristic in the next section.

We define the heuristic function f, as a weighted sum of g and h, so that their effect in the search can be manipulated: $f = (1 - \alpha)g + \alpha h$, where $\alpha \in [0, 1]$. Using $\alpha = 1$ corresponds to using the *min-size* heuristic. Using $\alpha = 0$ corresponds to a heuristic based only on the estimate of the height of the tree using the remaining variables. If we use $\alpha = 0.5$, then $2f$ provides a tight lower bound on the eventual height of the tree.

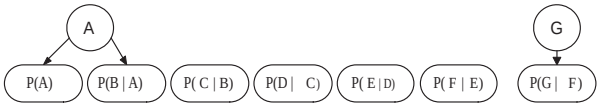
Because of the similarity to A* search, it is important to note that our approach



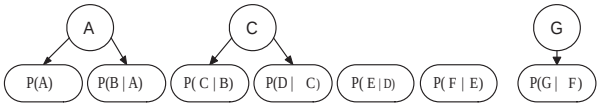
(a) Initialization. Note that there is one elimination tree for each CPT.



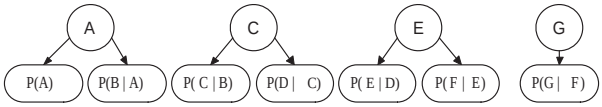
(b) After variable A is chosen.



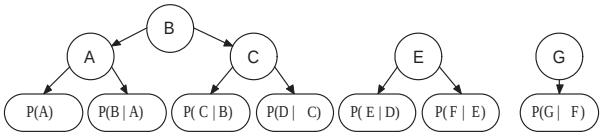
(c) After variable G is chosen.



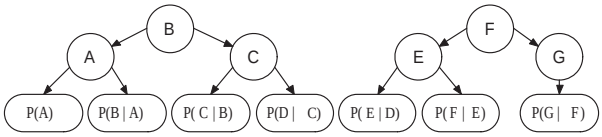
(d) After variable C is chosen.



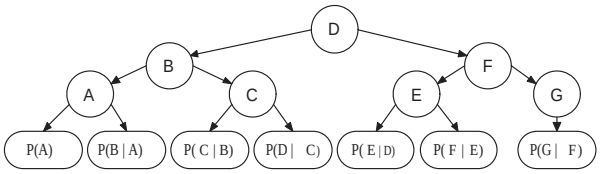
(e) After variable E is chosen.



(f) After variable B is chosen.



(g) After variable F is chosen.



(h) After variable D is chosen.

Figure 4.7: Elimination tree construction using the described heuristic.

is greedy: when selecting a variable to label a new node with, all alternatives but the one with the best estimate are discarded. Hence, the resulting ordering is not guaranteed to be optimal. Converting the greedy search to a best-first approach is a simple extension, which we have not fully explored.

In addition to using the *min-size* heuristic as a lookahead value, we also tested the *min-fill* heuristic as a lookahead value, since *min-fill* is typically preferred in classical variable-ordering applications over *min-size*. The problem with *min-fill* is that it counts edges, rather than variables, so an additive combination of g (which counts height in terms of a number of variables), and *min-fill* would not give a consistent estimate of total height. Furthermore, no simple setting of α can account for the difference in these measures. We resolve this problem by noting that if the number of remaining nodes to be marginalized is n, then the maximum number of necessary fill edges is $e = n(n-1)/2$. Solving for n gives $n = (1 + \sqrt{1+8e})/2$. This value derived from *min-fill* can be used as our lookahead value in the heuristic function f.

Finally, when selecting a node, it is very often the case that many variables have the same best f value, especially when selecting the first variables in the elimination order. Using the traditional methods, Darwiche and Huang recommend using the *min-fill* heuristic, breaking any ties with the *min-size* heuristic [34]. However, we found that even with tie-breaking procedures in place, there are still a large number of unresolved ties that have to be broken arbitrarily. To address this issue, we break these ties by choosing one of the best variables at random.

4.2.3 Evaluation

We compare the height of the elimination trees produced by the heuristics of Section 4.2.2 to those produced from converting balanced dtrees to elimination trees. We compare both the *min-size* heuristic, and the modified *min-fill* heuristic. This comparison is made using several well-known Bayesian networks from the Bayesian network repository [1], as well as the ISAC '85 benchmark (used in Darwiche and Hopkins [17] as test cases). We chose these networks for testing as they were the

Table 4.1: Heights of constructed elimination trees on repository Bayesian networks using the modified *min-size* heuristic for lookahead.

	DTree			Best-first search (values indicate α)										
	mf	*mb*	*hp*	*0.0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*
Barley	22	20	15	20	**14**	**14**	**14**	**14**	**14**	**14**	15	15	15	19
Diabetes	60	23	19	51	**18**	**18**	**18**	**18**	19	21	21	24	30	52
Link	46	43	48	148	**40**	**39**	**39**	**40**	**40**	**41**	**39**	**39**	**39**	47
Mildew	14	12	11	15	**9**	**9**	**9**	**10**	**10**	**10**	11	11	11	13
Munin1	23	22	26	42	**19**	**19**	**19**	**19**	**20**	**20**	**19**	22	22	23
Munin2	31	24	26	78	**16**	**16**	**16**	**16**	**17**	**17**	**19**	**20**	**22**	29
Munin3	27	21	24	79	**16**	**16**	**17**	**17**	**18**	**18**	**19**	**19**	**20**	28
Munin4	27	22	29	90	**17**	**17**	**17**	**18**	**18**	**19**	**20**	**21**	23	28
Pigs	26	25	24	48	**21**	**21**	**21**	**20**	**21**	**21**	**21**	**21**	**22**	26
Water	16	16	16	20	16	**15**	**15**	**15**	**15**	16	16	16	16	17

test networks in similar recursive decomposition applications, which makes the comparison between our results and other results easier to interpret.

Because all the heuristics employ random tie breaking, we show results as the mean of 50 trials for each configuration. The bold entries in the tables of results indicate where the mean height of the final tree using the search heuristics is superior to the best result from the dtree conversion methods.

Table 4.1 shows the results of the first comparison, using the benchmark Bayesian networks. In the first column (labeled mf), we show the mean height of elimination trees derived from a dtree constructed using the *min-fill* heuristic [15], without balancing. The second column (labeled *mb*) shows the height of elimination trees derived from balanced dtrees (using *min-fill* and *contract* [15]). The third column (labeled *hp*) shows the mean height of the elimination trees converted from a dtree constructed using hypergraph partitioning [17]. The remaining columns show the height of the elimination trees constructed using our modified *min-size* heuristic described above, for varying α values (the α values are given in each column header).

From this table, we can make a few observations. Considering only the dtree numbers, it can be observed that it is better to build an elimination tree from a balanced dtree, rather than an unbalanced one. Second, our results show that for constructing elimination trees from dtrees, there was no clear winner between using a dtree balanced using *contract* and a dtree constructed from hypergraph partitioning. Most notably, our modified *min-size* heuristic consistently outperformed the dtree

Table 4.2: Heights of constructed elimination trees on ISAC '85 benchmark circuits using the modified *min-size* heuristic for lookahead.

	DTree			Best-first search (values indicate α)										
	mf	*mb*	*hp*	*0.0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*
c432	42	41	47	74	42	**40**	**39**	**38**	**38**	**39**	**39**	**39**	**39**	42
c499	48	42	41	78	41	**39**	**39**	41	42	45	41	41	41	46
c880	56	52	54	125	**51**	**48**	**48**	**48**	**48**	**48**	**48**	**48**	**48**	56
c1355	50	50	46	124	**45**	**43**	**43**	**45**	**45**	**45**	**39**	**43**	**43**	52
c1908	74	72	85	196	88	83	81	77	76	74	74	74	74	79

based constructions, for α values between 0.2 and 0.5. The reductions in elimination tree height were between 1 and 8 variables for the networks tested. Considering that the complexity is exponential on the height of the tree, such a reduction is very significant.

Another interesting phenomenon occurring in the experiments was when $\alpha = 0$ and $\alpha = 1$. When $\alpha = 0$, the heuristic tries only to minimize the height of the current elimination tree being constructed. As demonstrated by the table, the elimination trees produced using only current cost are of a much poorer quality than when current cost and lookahead are used together. To explain this, recall that when only current cost is considered, there is potential for a partial tree to be constructed with many non-eliminated variables, which eventually results in a tall tree. On the other hand, when only lookahead costs are considered ($\alpha = 1$), then we see cases like the one in Figure 4.6. The best results appear for $\alpha \in [0.2, 0.5]$, which suggests that while using only the current height g creates very poor trees, the current cost should be weighted higher than the lookahead value h.

We also tested our heuristics using several of the ISAC '85 benchmark circuits, interepreting the circuits as DAGs, as these networks were also used when testing the quality of dtree construction methods [17]. Table 4.2 shows the results of this comparison. While the optimal α values are typically higher for these networks than the benchmark Bayesian networks, we see that the results are similar to the previous networks – the smallest means appear when $\alpha \in [0.1, 0.5]$. Our heuristic results in smaller trees than the standard *min-fill* algorithm, even after balancing the resulting dtree before converting to an elimination tree (except for a single network named

Table 4.3: Heights of constructed elimination trees on repository Bayesian networks using the modified *min-fill* heuristic for lookahead.

	DTree			Best-first search (values indicate α)										
	mf	*mb*	*hp*	*0.0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*
Barley	22	20	15	20	**14**	**13**	**13**	**14**	**14**	**14**	**14**	**14**	15	22
Diabetes	60	23	19	55	**18**	**18**	**18**	**18**	19	20	22	25	30	60
Link	46	43	48	152	**38**	**37**	**38**	**38**	**37**	**38**	**38**	**38**	**40**	46
Mildew	14	12	11	15	**9**	**9**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	14
Munin1	23	22	26	42	**19**	**18**	**18**	**19**	**20**	**19**	**20**	**20**	**21**	23
Munin2	31	24	26	81	**16**	**16**	**16**	**17**	**17**	**17**	**18**	**19**	25	31
Munin3	27	21	24	68	**16**	**16**	**17**	**17**	**18**	**19**	**19**	**20**	21	27
Munin4	27	22	29	81	**17**	**17**	**17**	**18**	**18**	**18**	**19**	**21**	22	27
Pigs	26	25	24	51	**19**	**20**	**20**	**20**	**20**	**20**	**21**	**21**	**22**	26
Water	16	16	16	**20**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	16

Table 4.4: Heights of constructed elimination trees on ISAC '85 benchmark circuits using the modified *min-fill* heuristic for lookahead.

	DTree			Best-first search (values indicate α)										
	mf	*mb*	*hp*	*0.0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*	*1.0*
c432	42	41	47	76	**39**	**38**	**37**	**38**	**38**	**38**	**39**	**39**	**39**	42
c499	48	42	41	78	**39**	**38**	**38**	**36**	**38**	**36**	**36**	**36**	**40**	48
c880	56	52	54	122	**49**	**47**	**45**	**44**	**44**	**45**	**45**	**45**	**46**	56
c1355	50	50	46	126	46	**44**	**45**	**41**	**40**	**39**	**39**	**39**	**43**	50
c1908	74	72	85	195	88	83	80	76	**70**	**70**	**70**	**71**	**68**	74

c1908).

Tables 4.3 and 4.4 show the results of using the modified *min-fill* measure as the heuristic to build elimination trees for the Bayesian networks and benchmark circuits, respectively. Again, the mean value of 50 trials is reported.

We can see from the results that *min-fill* outperformed the *min-size* heuristic as lookahead, for the example networks. The optimal α value appears to be lower (meaning that even less emphasis should be placed on lookahead). The results are more significant for the benchmark circuits, where the *min-fill* algorithm is superior to the dtree methods over all test networks (recall that *min-size* did not outperform the dtree methods for the *c1908* circuit.) Note that when $\alpha = 0$, the resulting elimination tree is poor, suggesting that some lookahead is always beneficial.

4.3 Indexing Improvements

The conditioning graph algorithm calculates CPT indices while variables are instantiated during the traversal of the *Query* algorithm. For each variable that has been observed or conditioned, the indices for its CPTs (linked through secondary pointers) are updated (Line 4 and 5 of the *Query* algorithm, Figure 3.8). These values must be unset once the child values have been calculated (Line 9 and 10 of the *Query* algorithm). This indexing occurs once for each time the node is visited; the number of times a node is visited is exponential in the depth of the variable in the elimination tree. This approach is simple to implement, but inefficient. We can dramatically improve the efficiency of indexing by precomputing some of parameters involved, at a small cost in terms of memory.

The function *index* takes a context over the variables of a CPT and returns a unique index for that context's entry in the CPT. We showed *index* in its Horner form (Equation 3.2), but we can also represent it as a linear function over its parameters. Let ϕ be a CPT in the Bayesian network, and let $dom(\phi) = \left\{X_{(1)}, X_{(2)}, ..., X_{(k)}\right\}$. We will assume without loss of generality that the subscripts of the variables in $dom(\phi)$ indicates the variable's ordering relative to the elimination tree: if $i < j$, then $X_{(i)}$ is an ancestor of $X_{(j)}$. Let $M_{(i)} = \prod_{j=i+1}^{k} m_{(j)}$ be the product of the cardinalities of all variables $\left\{X_{(i+1)}, ..., X_{(k)}\right\}$. Then:

$$index([x_{(1)}, \cdots, x_{(k)}]) = \sum_{i=1}^{k} x_{(i)} M_{(i)} \tag{4.1}$$

Assuming the cardinality of a variable never changes during inference, $M_{(i)}$ is a constant that can be calculated during the construction of the conditioning graph. There exists one of these constant values for each secondary link in the conditioning graph. We will refer to these values as *secondary scalar values*. Figure 4.8 shows the conditioning graph of Figure 3.7 with secondary scalar values attached to each link.

The commutativity of addition means that we can add the terms in Equation 4.1 in any order. Consequently, when a variable X at node N is observed, then for each

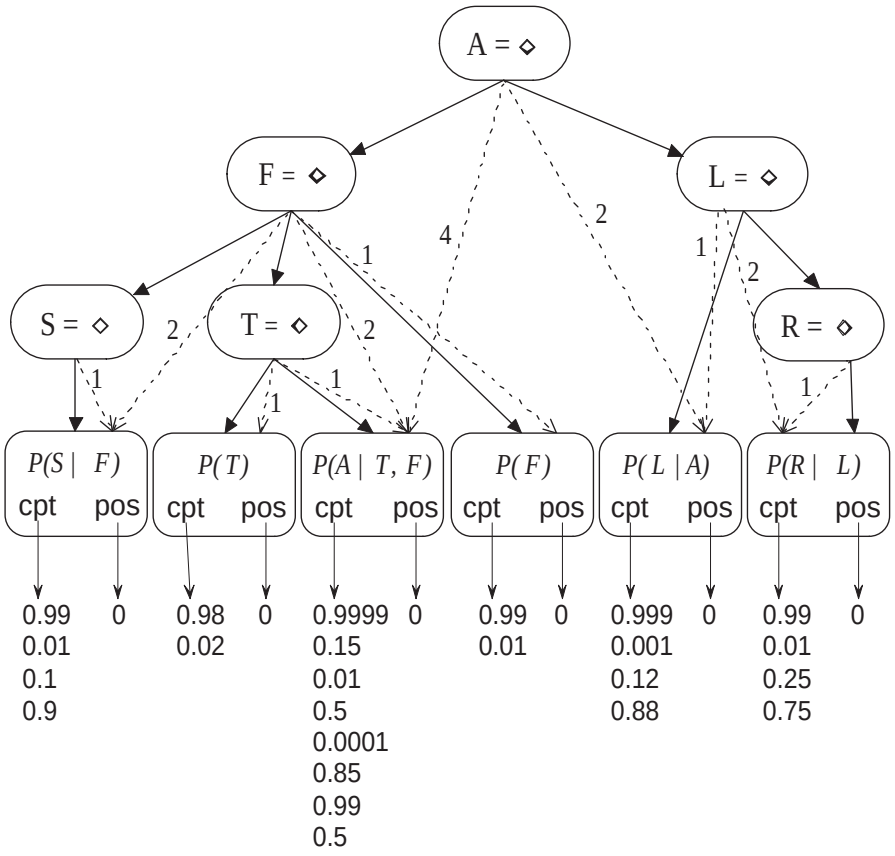


Figure 4.8: The conditioning graph, with the scalar values for each secondary link shown.

```
SetEvidence2(N, i)
1.    if i = N.value
2.      return
3.    diff ← i − N.value {⋄ = 0 in this equation}
4.    for each S′ ∈ N.secondary do
5.      S′.pos ← S′.pos + scalar(N, S′) ∗ diff
6.    N.value ← i
```

Figure 4.9: Algorithm for setting evidence, given that secondary scalar values are used.

leaf node S whose CPT domain includes X, we can adjust $S.pos$ accordingly using the secondary scalar value between N and S, *prior to any queries taking place*, or in the *SetEvidence* function. The new implementation, *SetEvidence2*, is defined in Figure 4.9. The function $scalar(N, S)$ represents the scalar value between a node N and a respective secondary child S.

The algorithm in Figure 4.9 adjusts the indexing values at each leaf node as soon as the value of an associated variable changes. Figure 4.10 gives the new algorithm for computing a probability from the conditioning graph. Notice that for an observed variable, the function does not use its secondary links to adjust the CPT indices, as it did in the first implementation (Figure 3.8, Lines 4,5,9, and 10).

As mentioned, one scalar value is created for each secondary link in the conditioning graph. In the worst case, the number of secondary links is quadratic on the number of nodes in the network, but typical Bayesian networks are sparse with respect to edges. However, in cases where memory does not permit the inclusion of these values, one can always revert back to the original algorithm.

The complexity of inference in conditioning graphs is a function of the number of recursive calls made. Since improving indexing does not reduce this number, the savings provided by the better indexing scheme is asymptotically 'hidden' in the base of the complexity term. However, the actual savings provided can be substantial, especially in the lower nodes of the tree. Let $s = |N.secondary|$ be the number

```
Query2(N)
1.  if N is a leaf node
2.    return N.cpt[N.pos]
3.  else if N.value <> ◇
4.    Total ← 1
5.    for each P' ∈ N.primary while Total > 0 do
6.      Total ← Total * Query2(P')
7.    return Total
8.  else
9.    Total ← 0
10.   for i ← 0 to N.m − 1 do
11.     SetEvidence2(N, i)
12.     Total ← Total + Query2(N)
13.   SetEvidence2(N, ◇)
14.   return Total
```

Figure 4.10: Algorithm for querying, given that secondary scalar values are used.

of secondary links for node N, and $m = N.size$. In the original algorithm specification (Figure 3.8), adjusting CPT indices upon each visit to node N required $2ms$ multiplications and ms additions. Under the new indexing scheme, the number of multiplications is reduced by a factor of 2 for unobserved nodes. For observed nodes, no arithmetic operations are required for CPT indices (they were already performed prior to query). This means a potentially exponential reduction in the number of arithmetic operations, especially when the number of observed variables is low. Hence, the savings at each node is a function of that node's depth in the tree, and will be substantial for deep nodes.

4.4 Unobserved Leaf Variables

The following discussion focuses on an optimization targeted at the leaf variables of a Bayesian network. We will refer to variables in the Bayesian network with no children as *leaf variables*, and childless nodes in a dtree, elimination tree, or conditioning graph as *leaf nodes*. We do this to avoid confusion, as both the Bayesian network and its compiled forms have leaves.

In Recursive Conditioning [15], dtree algorithms never marginalize a leaf variable. This is because a leaf variable can never be part of a cutset that partitions the network, since it has no outgoing arcs. However, this explanation leaves out what happens at a leaf variable's corresponding CPT node in the dtree. For instance, consider Figure 4.3. If the dtree does not marginalize a leaf variable E, then E never receives a value, and the generated context at leaf node $P(E|D)$ is incomplete; no value of E exists. However, if the algorithm did in fact iterate over the values of E, it would simply return 1, since $\sum_E P(E|D) = 1$ for any value of D. Hence, the recursive conditioning algorithm accounts for this with a special case when the CPT of a leaf variable from the Bayesian network is reached: if the corresponding leaf variable does not have a value in the context, the return value of that node is 1. If it does have a value, then it performs a lookup into the CPT, as before.

This technique can have an exponential reduction on runtime, and therefore it would be beneficial to do something similar in a conditioning graph. However, the internal nodes represent an input for evidence for the user. Hence, we would like to explicitly represent each variable. As well, any 'special cases' in the algorithm should be trivial, so that the small size of the algorithm is not affected.

Let V be a leaf variable in the Bayesian network. If node N is the primary parent of the leaf node that contains V's CPT, and V is the variable that labels N, then we can exploit leaf variables in much the same way as dtrees. The conditioning graph of Figure 3.7 and the elimination tree of Figure 4.3 both meet this property. We first show how this exploitation works, we will then consider how to construct the conditioning graph to ensure these properties exist.

```
Query3(N)
 1.  if N is a leaf node
 2.    return N.cpt[N.pos]
 3.  else if N.value <> ◇
 4.    Total ← 1
 5.    for each P' ∈ N.primary while Total > 0 do
 6.      Total ← Total * Query2(P')
 7.    return Total
 8.  else
 9.    if N.isLeaf
10.      return 1
11.    Total ← 0
12.    for i ← 0 to N.m − 1 do
13.      SetEvidence2(N, i)
14.      Total ← Total + Query2(N)
15.    SetEvidence2(N, ◇)
16.    return Total
```

Figure 4.11: Algorithm for querying, given that leaf variable nodes are labeled.

Let N be the node in the conditioning graph labeled with variable V's CPT. If N' is the parent of N and V is the variable labeling N', then marginalizing V at N' would produce a value of 1, as mentioned. Hence, when the node of a leaf variable is reached, we can immediately return the value 1 if the variable is not observed. This system requires that such a node be labeled; let *N.isLeaf* be a boolean value that denotes that node N contains a leaf variable V from the Bayesian network, and that N contains one primary child, which is the node containing V's CPT. Given this labeling, the conditioning graph algorithm can be easily modified to accommodate the change. Figure 4.11 shows the *Query* algorithm complete with the modifications (Line 09 and 10).

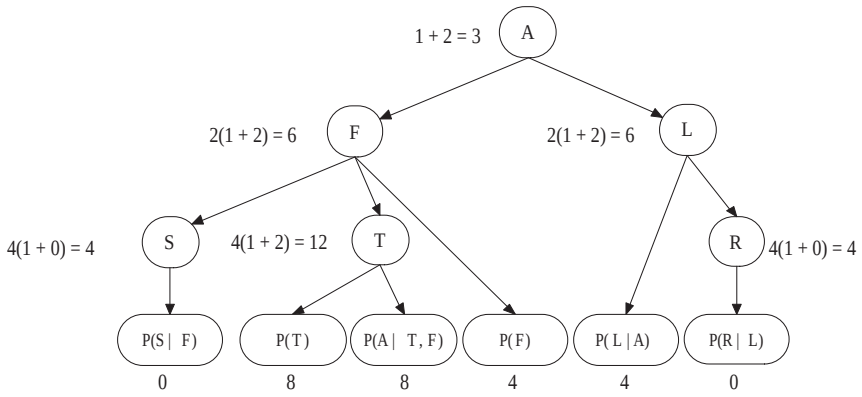


Figure 4.12: The *Fire* elimination tree. Number of recursive calls to each node is shown beside (or below) the node.

Again, this optimization requires that the leaf variables be positioned directly above their corresponding CPTs in the conditioning graph. We can ensure that this property holds by selecting these variables first in the elimination ordering during construction. The *min-fill* heuristic actually produces this behaviour in many cases, although there is no guarantee.

The leaf-variable optimization is very simple and can be very effective. It effectively reduces the number of unobserved variables along each path containing a leaf variable by 1, which can mean a speedup equivalent to the cardinality of the leaf variable. Consider Figure 3.10(a), which shows the number of recursive calls made to each node with no evidence. The variables *Smoke* and *Report* are leaf variables and are directly above their corresponding CPTS, and hence will always return the value 1 given no evidence. However, by exploiting leaf variables, the number of recursive calls is also reduced at the *Smoke* and *Report* nodes themselves, since we never marginalize these variables (and thus they never perform their respective "self" calls). Figure 4.12 shows the number of recursive calls to each node reduces from 91 to 59, a savings of almost 40% for this example. The actual savings will depend on the structure of the network (how many leaf variables exist), and what the context is (whether the leaf variables are observed or not).

The optimization of this section can be explained in terms of barren variables (Section 2.1), as an unobserved leaf variable in a Bayesian network is always barren,

and can therefore be ignored during computation. In Section 5.3.2, we will demonstrate methods for ignoring all barren variables, which would make the optimization of this section obselete. However, ignoring all barren variables requires significant extra runtime costs, as well as extra memory.

4.5 Summary

This chapter described general optimizations of the conditioning graph, to improve the time required for calculating probabilities. By general optimizations, we refer to improvements that affect the conditioning graph irrespective of the application. In the following chapter, we will consider application-specific advantages.

The first improvement demonstrated techniques for building good elimination trees, from which we can construct conditioning graphs. Since the time complexity of computing over an elimination tree is a function of its height, a shallow, balanced elimination tree is desirable. We described a linear-time transformation from dtrees to elimination trees that guarantees the complexity of the two structures are the same. We developed two new heuristics for directly building elimination trees, extended from traditional heuristics for developing variable orderings in Bayesian networks. We showed that in the example networks, the elimination trees developed from these heuristics are typically smaller than those converted from dtrees. The results of experiment show that the proposed heuristics are actually preferable to other methods for construction when the time complexity is a function of height (no caching). This applies not only to elimination trees, but to dtrees as well.

The second optimization improved the efficiency of indexing in the CPTs of the conditioning graph. This optimization required a simple extension to the original algorithm which is consistent with the original goal of conditioning graphs: easily implementable, making them universally portable. The optimization avoids repeat calculation, saving an exponential number of arithmetic operations for a given query, and these savings can be realized across queries in cases where the evidence remains the same. This optimization increases the storage requirements of each secondary

arc by only a small constant factor.

The third optimization demonstrated a method for ignoring the leaf variables of the Bayesian network when they were unobserved. The optimization required a simple boolean label for each internal node, and a small addition to the original inference algorithm. The optimization can potentially reduce the height of the tree by an entire layer, providing exponential speedup, in cases where the leaf nodes are (mostly) unobserved. As with the indexing optimization, the space required for this optimization is very small, adding a small constant factor to node storage.

Chapter 5 continues to explore optimizations to the conditioning graph model. We will consider application-specific optimizations, which are independencies that arise given certain evidence and query contexts. The chapter demonstrates how to exploit evidence variables, barren variables, and d-separated variables, both offline and online. We demonstrate that these optimizations allow inference operations over conditioning graphs to have feasible runtimes in comparison to exponential space algorithms such as JTP and VE, while maintaining their conservative space requirements.

Chapter 5

Application-specific Optimizations

5.1 Introduction

Inference in Bayesian networks allows the calculation of posterior probabilities while considering only essential information. Any information deemed irrelevant to the current query is ignored by certain inference algorithms (such as Variable Elimination). This can provide an enormous efficiency gain in application, both space and time-wise. Finding irrelevant information is linear on the size of the network model, making it fast in comparison to inference.

Because precompiled structures, like conditioning graphs and junction-trees, must be general enough to allow any query, they do not inherently exploit the irrelevance of certain information for a given context. In this chapter, we demonstrate how to exploit such independencies in conditioning graphs. We show that with a small amount of additional memory, we can achieve exponential speedup in many cases.

We categorize the optimizations of this section into two classes. The first class includes optimizations that we can apply at compile-time. The more computation performed at compile-time, the less that we require at run-time. We show how to exploit sensor variables (variables that are always observed), resulting in a substantial decrease in the required computation over the network. We also show how knowledge of query and evidence variables in advance allow offline partial elimination, increasing the performance while potentially decreasing space requirements.

The second class of optimizations are run-time optimizations. We demonstrate methods for ignoring information that is irrelevant because of the current context, which includes barren variables and d-separated variables. We also show how to

generalize one of the compile-time techniques for exploiting sensor variables to non-sensor variables at runtime. Given these methods, we show that conditioning graphs exhibit reasonable time complexity when compared to VE, while still requiring only $\mathbf{O}(n \exp(f))$ space.

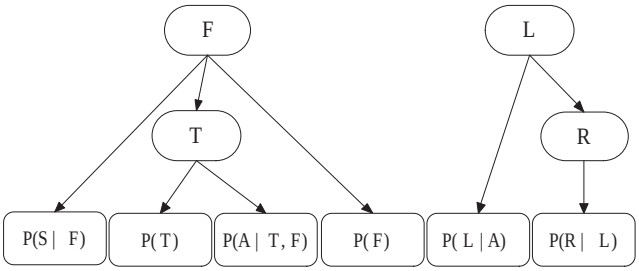
5.2 Compile-time Optimizations

A major advantage of conditioning graphs is their offline compilation (Chapter 3). Much of the work that is normally associated with Bayesian network inference is performed during the compilation step, and can therefore be ignored in terms of the runtime requirements of the algorithm. In this section, we demonstrate techniques that exploit application-specific knowledge to allocate more computation from runtime to compile-time, thereby increasing the efficiency of the online conditioning graph operation.

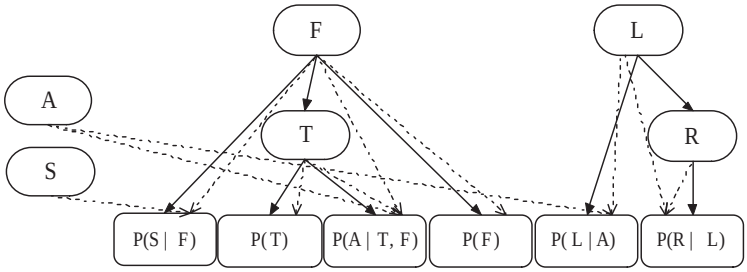
5.2.1 Sensor Models

It is well known that one can condition a Bayesian network on the evidence before performing inference [65]. This reduces network connectivity, resulting in smaller cutset widths, and eliminates the evidence nodes from the CPTs, resulting in fewer marginalizations. If we know that some set of variables will always be observable, we can likewise modify the conditioning graph to be more efficient. This is a realistic situation: in any application, there typically exists at least a small subset of variables that are always observable. Examples of these include monitor output in medical patient monitoring, and sensor readings in car diagnosis. We refer to variables that can always be observed as *sensor variables* [18].

Let $\mathbf{E}$ be the set of sensor variables for a Bayesian network. We construct the elimination tree over the Bayesian network by creating internal nodes for all variables except those in $\mathbf{E}$. All of the CPTs are included in the tree. A conditioning graph is constructed from the elimination tree as before, with secondary arcs from each internal node to the appropriate leaf nodes. The variables in $\mathbf{E}$ also have secondary



(a) An elimination tree in which $Alarm(A)$ and $Smoke(S)$ are never marginalized.



(b) Adding evidence nodes for *Alarm* and *Smoke*.

Figure 5.1: The new conditioning graph, which removes primary arcs from the sensor variables. Note that for space consideration, we use the CPT notation, rather than listing the array of values explicitly.

arcs to their respective leaf nodes, but they are not connected to the tree structure with any primary arcs.

Considering the *Fire* model (Figure 3.7), suppose it is known in advance that the state of the fire alarm will always be observable, as well as whether or not there is smoke present (both are easily accomplished using sensors). Hence, our set of sensor variables is $\mathbf{E} = \{S, A\}$. Following the process outlined in the previous paragraph, we construct a tree that does not include nodes for variables S or A. See Figure 5.1(a). A conditioning graph is constructed from this elimination tree as before, with secondary arcs from each internal node to the appropriate leaf nodes. Notice that the variables in $\mathbf{E}$ are given secondary arcs pointing to the appropriate leaf nodes, but they are not connected to the tree structure with any primary arcs. Figure 5.1(b) shows the resulting structure.

To order the entries in the CPTs, an ordering is selected for the sensor variables, and the CPT entries are ordered first on their sensor variables, followed by those

in the elimination tree. This gives us a total ordering over the variables, which, in addition to providing an order for CPT entries, also allows us to calculate secondary scalar values, if memory permits.

There are definite benefits to this separation of the evidence nodes from the conditioning graph. Leaving $\mathbf{E}$ out of the elimination tree may result in several distinct trees, each of which is smaller than if they were included. Computing $P(x_q, \mathbf{e})$ only requires processing the component containing X_q in its nodes. Thus, even though our conditioning graph is static at run-time, we are able to "prune" away irrelevant parts of the model during compilation. Note that this requires a pointer from each variable X_q to its corresponding elimination tree, but these pointers require only linear space to store. There are other advantages. Reducing the conditioning graph by leaving out the observable variables may reduce its height by producing subtrees, which can bring about exponential speedup when computing probabilities. Considering our example, removing the sensor variables from the original graph reduces the maximum height from 3 to 2, and the average depth of each CPT from 2.67 to 1.5. Plus, as long as the evidence remains the same, we need only process the relevant elimination tree to handle multiple queries.

5.2.2 Query Variables

In Variable Elimination, it is well known that eliminating barren variables can improve the time it takes to process a query. Also, any nodes in the Bayesian network that are d-separated from the query can be removed. We can perform similar pruning in the conditioning graph if we know in advance a subset of *hidden* variables that will never be queried or observed (runtime network pruning will be discussed in the next section).

Recall that when our conditioning graph is disconnected (has multiple subtrees), then calculating $P(x_q, e)$ requires computing only over the subtree containing X_q. If it is known in advance which variables will be queried, then the separation induced by the sensor nodes may produce subtrees which are never computed over. From our previous example, if we knew that variables *Report* and *Leaving* would never

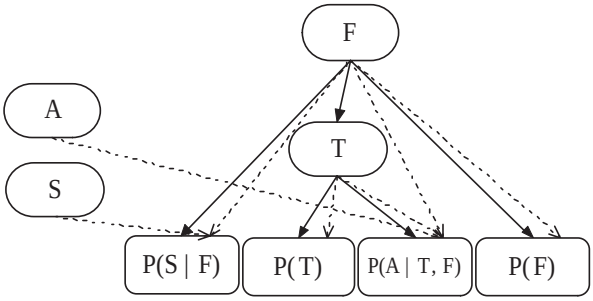
be queried or observed, then that portion of the network need not even be stored. Figure 5.2(a) shows the new structure.

Knowing the query variables in advance allows us to perform some computation in advance (at compile-time), saving us future computations while the system is live. For instance, if an internal node in a conditioning graph has several leaf nodes, the distributions can be multiplied at compile time, and the single distribution made the only child of the node. This will reduce the number of multiplications during inference, but has the potential to increase the space requirement of the problem. Thus it should only be performed if this increase in size is acceptable. On the other hand, it is possible that this operation may decrease the space required to store the conditioning graph.

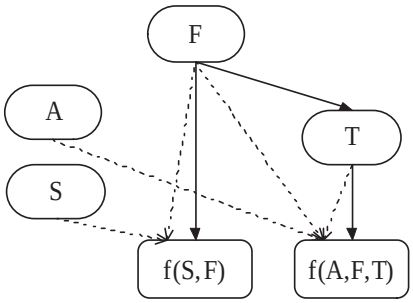
From our previous example, we see that the internal node associated with *Tampering* has two leaf nodes, whose CPTs correspond to $P(A|T,F)$ and $P(T)$. Multiplying these two CPTs produces a factor over $\{A,T,F\}$, with 8 values. This operation does not add to the space requirements (in fact, it reduces them). Similarly, the node for *Fire* has two leaf nodes that can be multiplied with similar effect. Figure 5.2(b) shows the conditioning graph after these two optimizations are performed. Note that the size of the model and the number of computations necessary has been reduced.

We can take this optimization one step further considering that we know of variables that will never be observed or queried. If a subtree in the elimination tree contains only variables that will never be queried or observed, then we can compact that subtree into a single leaf node at compile time. This amounts to doing partial elimination, before we condition, and storing an intermediate distribution, rather than all CPTs from the original network. Once again, this step has the potential to increase the space requirements of the conditioning tree. However, we can calculate the size of the new leaf node in advance, without actually performing the computation. This allows us to decide beforehand whether such partial elimination is acceptable given our current size restrictions.

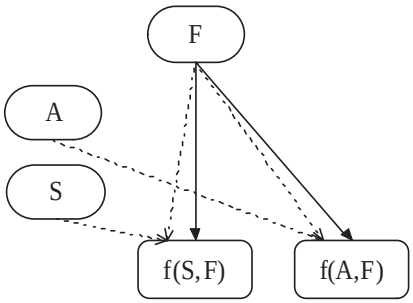
Continuing with the example, suppose that the need to query the *Tampering*



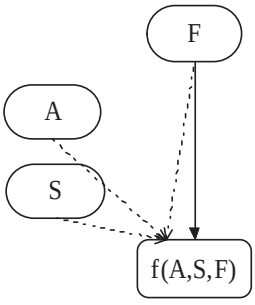
(a) The conditioning graph, leaving out *Report* and *Leaving*.



(b) The conditioning graph, with leaf nodes for each internal node compacted.



(c) The conditioning graph, with *Tampering* marginalized out.



(d) The final conditioning graph.

Figure 5.2: Optimizing the conditioning graph.

variable is now eliminated, and assume that it will never be observed. Hence, we can multiply all of its children (there's only one in this example), and marginalize out the *Tampering* variable. Figure 5.2(c) shows the system after we perform this step. Note that this operation further reduces the height of the tree. As well, the *Fire* variable now has two leaf nodes, that can be compacted without increasing the space complexity. Figure 5.2(d) shows the final product, an extremely small, efficient version of the original problem. In fact, we have reduced it to a simple lookup, given the values of the evidence and query. Stated another way, the problem has been reduced to an explicit representation of a joint probability distribution over the query and evidence variables. The significance of this is that this happens automatically - the algorithm has the ability to determine when storing a joint probability distribution is more efficient than storing a factorized version in some cases. Note that such a reduction is not always possible, but removing sensor variables from the elimination tree and performing partial elimination can reduce considerable portions of the network given the right variable ordering.

5.3 Runtime Optimization

The methods of the previous section allow the conditioning graph structure to be optimized at compile-time, given that we know in advance our evidence variables and/or query variables. However, there are some limitations to these compile-time steps. While it is reasonable to assume that we will know a set of sensor variables and query variables in advance, these sets may change while the system is online. As these sets change, so does the relevant portion of the graph that we need to compute over. That is, we can further optimize based on new information. However, further optimization requires that we perform these optimizations at runtime. This section is devoted to such runtime optimizations, that take advantage of the application-specific information we considered in the previous section, only in a dynamic fashion.

5.3.1 Hoods

In Section 5.2, nodes containing sensor variables (variables that are always observed) were treated independently of the non-sensor variables. This means that we can apply evidence to the conditioning graph prior to and independent of any query. This also means that the original elimination tree decomposes into a set of smaller elimination subtrees. Each non-sensor variable is d-separated from all nodes outside of its subtree. To query a variable X, we call *Query* on the variable containing X.

This model for handling evidence is sufficient for cases variables exist that are always observed. In the event that one of these variables becomes unobservable, some of the independence assumptions between groups of nodes becomes invalid. As an example, if the sensor variable *Alarm* in our previous tree were to fail, then the trees containing *Leaving* and $Fire$ are no longer independent of each other. The node containing the sensor variable must somehow be reincorporated into the tree, requiring either (a) a special case algorithm or (b) a recompilation.

In addition, suppose that a node is not actually a sensor variable, but has a high probability of being observed. It would be unwise to declare it as a sensor variable, in the event that it becomes unobserved. However, if the node is observed during a particular query, then it would be nice to take advantage of this and include it in the sensor variables, partitioning its subtree further and reducing the complexity of inference over the variables of this subtree.

Rather than maintain the sensor nodes separately from the tree, we incorporate them into the original tree structure (just as if they were not sensor variables at all). To ensure that the sensor variables are still partitioning the structure into independent subtrees, we require that they comprise the top of the tree. To obtain this structure, we simply construct the tree as before (using the modified VE algorithm). However, this time we sum out all of the variables, making sure to sum out the sensor variables last.

We will refer to the *hood* of the elimination tree/conditioning graph as the topmost set of connected functioning sensor nodes. The hood of the tree can be defined

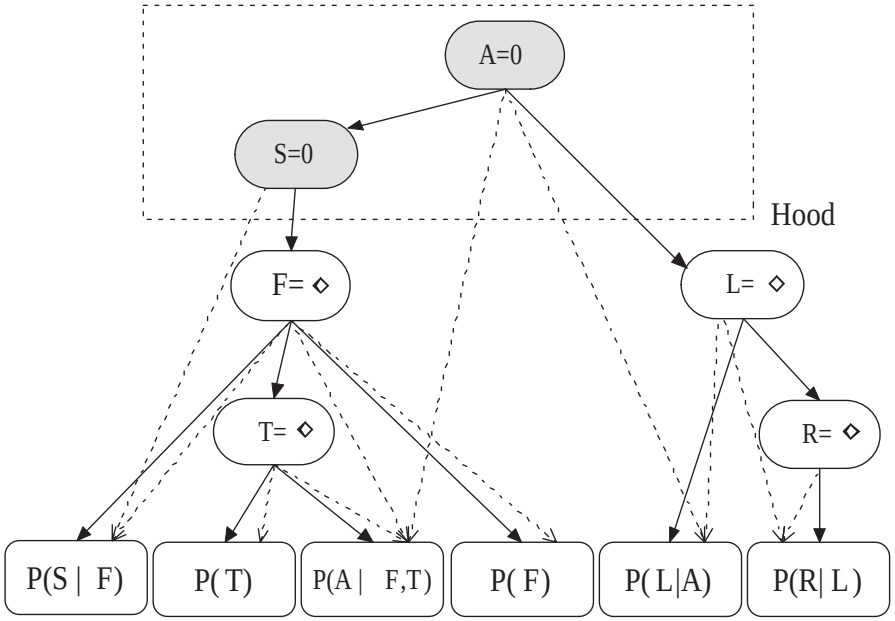


Figure 5.3: The hood of the *Fire* example, given sensor variables *Smoke* and *Alarm*.

recursively:

- The root of the elimination tree belongs to the hood if its variable is observed.
- Any node in the elimination tree belongs to the hood if its parent belongs to the hood, and its variable is observed.

Figure 5.3 shows the Fire network given sensor variables *Smoke* and *Alarm*. Note that the subtrees (the trees rooted outside of the hood) have not changed.

Maintaining the sensor variables as a hood eliminates the need for a special case algorithm if a sensor variable fails. In the event of a sensor failure, the hood is reassessed, making previously independent subtrees dependent again automatically. And since all of the sensor nodes are already a part of the tree, this requires no reincorporation step.

The set of hood variables can be represented by a boolean value at each node, which we refer to as *N.hood*. *N.hood* will be true if the variable belongs to the hood, and false otherwise. Both adding and removing nodes from the hood can be accomplished with a depth-first traversal. However, if each node stores a pointer

```
SetEvidence3(N, i)
1.  SetEvidence2(N, i)
2.  if i ≠ ⋄ AND N.hood = false AND (N.parent = null OR N.parent.hood = true)
3.    SetHood(N, true)
4.  else if i = ⋄ AND N.hood = true
5.    SetHood(N, false)

SetHood(N, h)
1.  N.hood = h
2.  for each P' ∈ N.primary s.t. P'.value ≠ ⋄ do
3.    SetHood(P', h)
```

Figure 5.4: Algorithm for setting the evidence, incorporating changes to the hood.

to its parent ($N.parent$), then we can dynamically add a node to the hood of the network, if the parent of that node belongs to the hood. This makes the hood of our network fully dynamic with the incoming evidence, therefore, we can amalgamate the algorithm for setting evidence and maintaining the hood. When a node receives an observed value, it sets its value and checks to see whether it is part of the hood. If it is not, and its parent is a part of the hood (or it is the root of the conditioning graph), it adds itself to the hood, updates the index at its secondary children, and calls for each child to test whether it's part of the hood. Each observed child recursively adds itself to the hood, sets its secondary children, and calls its observed children to add themselves. Conversely, when a node belonging to the hood becomes unset, it removes itself from the hood, and recursively calls its children to do the same. Figure 5.4 shows a new implementation of *SetEvidence* that maintains the *hood* values in the nodes. The *Query* algorithm remains unchanged.

The hood variables dynamically separate the conditioning graph into smaller subgraphs. A changing hood requires that the mapping from variable to the subgraph that contains it must change as well. When a variable whose node has more than one child is added to the hood, each of the children nodes become an independent

subtree. When a variable is removed from the hood, all of its children subtrees must be amalgamated into one tree. Hence, in each case, the mapping of variables to these subtrees must change. There are two ways to change the mapping, depending on the application. If the model is being queried often, then when the hood changes, you may wish to perform a DFS traversal, changing the mapping for each variable as you go. However, if you are querying the network infrequently, and the set of observed variables is changing often, then performing a DFS traversal each time may incur a lot of needless work (for instance, if no queries take place for a particular configuration of evidence). In this case, it may be better to traverse along the parent pointers of each node to the root of the variable's respective subtree.

The benefits of the hood optimization largely depends on the context in which it is used. In applications where many of the variables are observed, implementing a hood over a conditioning graph can reduce the average query size substantially, through the partitioning of the tree into subtrees. However, in cases where very few observations are made, the extra overhead incurred by maintaining a hood may outweigh its benefits.

5.3.2 Relevant Variables

In Chapter 2, we examined information in the Bayesian network that was irrelevant given certain query and evidence combinations. Observing a variable has the effect of removing its outgoing arcs from the Bayesian network. A *barren* variable and all of its incident arcs can be pruned from the Bayesian network. Given these operations, any variable that is disconnected from the query variable(s) is considered to be d-separated from the query, and can be ignored from the computation. Finding and computing over only the relevant information for a particular query is the topic of this section.

Finding barren and d-separated variables requires traversal through the Bayesian network, but the conditioning graph model as presented so far does not store the Bayesian network in a convenient manner for this. We first discuss ways to represent the Bayesian network structure in a conditioning graph. Once this is established, we

can use this structure to determine barren and d-separated variables.

In the following discussion, we will be discussing Bayesian networks and conditioning graphs, both of which use the notion of parents and children. When referring to Bayesian network components, we will prepend the component name with *network*. For example, the *network parent* of a variable are its parents in the Bayesian network. In contrast, the parent of a node refers to the node's parent in the conditioning graph.

For representing the Bayesian network structure in the conditioning graph, we consider two possibilities:

1. At each node, store two separate sets of pointers that correspond to the arcs in the original Bayesian network. That is, node N storing variable V would have two sets, *pa* and *ch*, that point to the nodes containing V's network parents and network children, respectively.

2. Make the secondary arcs bi-directional, so that they can be traversed from leaf node to internal node. We define the *root arc* of a node in the conditioning graph as follows: let N be an internal node labeled with variable X_i. Then the root arc of N is the secondary arc pointing to the leaf node labeled by $P(X_i|\Pi_i)$. This is also the root arc for the leaf node labeled by $P(X_i|\Pi_i)$. Note that every node in a conditioning graph has exactly one root arc. Figure 5.5 shows the conditioning graph of Figure 5.3 with the root arcs highlighted.

If bi-directional arcs are used, then the network parents and children of variable V can be found as follows: Let N be the node labeled with variable V. The nodes containing the network parents of V are found by traversing the root arc of N to N', and then traversing the non-root arcs of N'. Conversely, the network children of V are found by traversing the non-root arcs of N, followed by the root arc of each of those nodes. As an example, to find the network parents of variable $Alarm(A)$ in Figure 5.5, we first traverse the root arc from A, which takes us to the node labeled by $P(A|T,F)$. Traversing the non-root arcs from this node takes us to the nodes labeled by T and F, which are the network parents of A. Likewise, to find

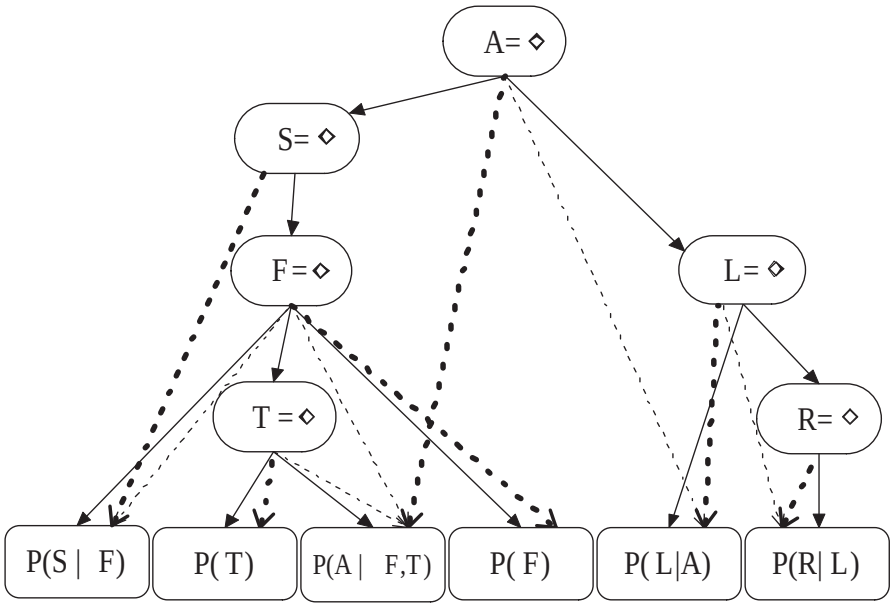


Figure 5.5: The *Fire* conditioning graph of Figure 5.3. Root arcs are shown with bold dotted lines.

the network children of variable F, we first traverse the non-root arcs from F, which leads to the nodes labeled with $P(S|F)$ and $P(A|T,F)$. Traversing the root arcs from these nodes leads to the nodes labeled with S and A, which are the network children of F.

Using a separate set of pointers to represent the Bayesian network structure in a conditioning graph is more intuitive, and require only one step to traverse to a neighbour (rather than the two step process of traversing to a leaf node first). However, including these pointers requires more space than making existing secondary arcs bidirectional if the number of arcs exceeds the number of nodes in a Bayesian network. For simplicity, we will use a separate set of pointers in our algorithms, but these algorithms are easily modified to use the second option if space is limited.

Now we consider the problem of exploiting barren and d-separated variables in the conditioning graph. There exist several algorithms for determining the information that is relevant to the query [30, 61]. We use a variant of these algorithms, that finds barren variables and d-separated variables separately: non-barren variables are maintained in response to evidence changing, and the remaining relevant variables

are determined at query-time.

The barren variables of a Bayesian network can be identified recursively: a variable in a Bayesian network is barren if (a) it is not observed and not part of the query and (b) either it is a leaf node, or all of its children are barren. For our algorithm, we maintain the collection of non-barren variables dynamically, as follows: whenever a barren variable becomes observed (or part of a query), then it becomes non-barren, and notifies its network parents of its non-barren state. This process continues in a recursive manner. Conversely, when a non-barren variable becomes unobserved, it checks whether or not its children are all barren. If they are, it becomes barren, and notifies its parents of its barren-ness. To accomplish this in a timely fashion, each internal node in the conditioning graph maintains an integer, *nonbarren*, that represents the number of nonbarren network children that variable has. When a variable becomes non-barren, it notifies its network parents, which update their *nonbarren* status by incrementing it. The opposite process occurs when a non-barren node becomes barren. A variable is barren if it is not observed and its *nonbarren* value is 0. Figure 5.6 shows *SetEvidence4*, our new evidence entry method that maintains barren variables. Note that *SetEvidence4* is called whenever the observed value of a variable changes, independent of any query.

The relevant information to a query variable (or set of query variables) is the subgraph of the Bayesian network that is still connected to the query variables after pruning barren variables and the arcs emitting from observed variables. We can identify this information by performing a graph traversal, beginning from each query variable and being careful not to traverse an arc that has been pruned from the network. This means that if a node is visited during the travesal, then it is not disconnected from the query and is therefore relevant. The rules of the traversal can be defined as follows:

1. We may not traverse to a barren variable, as such a variable is considered to be pruned for this particular query.

2. We may not traverse an arc whose origin is an observed node.

```
SetEvidence4(N, i)
1.  SetEvidence2(N, i)
2.  if i ≠ ◇
3.     ResetBarren(N)
4.  else
5.     SetBarren(N)

ResetBarren(N)
1.  if N.barren = true
2.     N.barren ← false
3.     for each Pa ∈ N.pa do
4.        Pa.nonbarren ← Pa.nonbarren + 1
5.        ResetBarren(Pa)

SetBarren(N)
1.  if N.barren = false AND N.nonbarren = 0 AND N.value = ◇
2.     N.barren ← true
3.     for each Pa ∈ N.pa do
4.        Pa.nonbarren ← Pa.nonbarren - 1
5.        SetBarren(Pa)
```

Figure 5.6: Algorithm for setting the evidence, maintaining labeling of barren nodes.

```
SetRelevant(N)
1.  for each node X in the graph
2.    X.relevant ← false
3.    X.active ← false
4.  MarkRelevant(N,N)

MarkRelevant(N, Q)
1.  N.relevant ← true
2.  MarkActive(N.root)
3.  for each P ∈ N.pa s.t. P.barren = false AND P.relevant=false AND P.value= ⋄
4.    MarkRelevant(P, Q)
5.  if N = Q OR N.value = ⋄
6.    for each C ∈ N.ch s.t. C.barren=false AND C.relevant=false
7.      MarkRelevant(C, Q)

MarkActive(N)
1.  N.active ← true
2.  if N.parent.active = false
3.    MarkActive(N.parent)
```

Figure 5.7: The *SetRelevant* algorithm, which marks the active part of the conditioning graph for processing a particular query.

These simple rules allow us to write a depth-first search algorithm for marking the relevant nodes. This algorithm, *SetRelevant*, is given in Figure 5.7. To identify relevant variable, a boolean value *relevant* is attached to each node, and is given the value *true* for each graph node which contains a relevant variable.

Define a *relevant CPT* to be a CPT that is to be included in the computation. The set of relevant CPTs is defined as all CPTs $P(X_i|\Pi_i)$ such that X_i is not barren and at least one variable from the set $\{X_i\} \cup \Pi_i$ is relevant. We will use the keyword relevant to also indicate CPTs that are relevant to the problem. If a subtree contains no relevant CPTs, then there is no point to traversing that subtree. Given a relevant CPT, we will define the path from the root to that CPT's node in the conditioning

graph as an *active path*. Only the active paths in the conditioning graph need be traversed; all other paths can be ignored. An internal node in a conditioning graph is *active* if it is part of at least one active path in the conditioning graph.

We identify each active node in the conditioning graph by setting a value *active=true*. We use the *MarkActive* algorithm in Figure 5.7 to mark the active nodes in the graph concurrently with identifying relevant variables. When *MarkRelevant* marks a relevant variable at node N, it uses the root arc from that N (denoted $N.root$) to call *MarkActive* on that variable's CPT. Note that *MarkActive* also requires that each node N have a pointer to its parent node (the $N.parent$ value discussed previously).

As an example of this marking system, consider the conditioning graph in Figure 5.8, taken from Section 5.3.1. Before any query or evidence nodes are specified, all nodes are barren, and therefore irrelevant. Suppose we observe that people are leaving ($L = 1$) but no report has been generated ($R = 0$). Figure 5.9 shows the new state of the graph. Note that only S is still barren; we need to know the set of query variables to decide which other variables, if any, are irrelevant. If we were to make F a query variable, the search labels both S and R irrelevant, since S is still barren, and R is d-separated from the query by L (Figure 5.10). Finally, the active nodes are shown in Figure 5.11. Notice that S is active even though it is irrelevant, due to an active child.

Given that the active and relevant nodes have been marked in the conditioning graph (that is, *SetRelevant* has been called on the query node), *Query3* in Figure 5.12 shows the new query algorithm. The new query algorithm only traverses the active part of the network. It only conditions over relevant nodes. Each node now additionally stores pointers to the nodes containing its network parents and children, and maintains *nonbarren*, *relevant*, and *active* flags. These additions cumulatively contribute a storage requirement that is linear on the number of nodes in the network.

It is well known that pruning away irrelevant information has the potential for enormous savings in Bayesian network inference [30, 61]. The same statement applies to conditioning graphs. We have seen that the running time of inference is

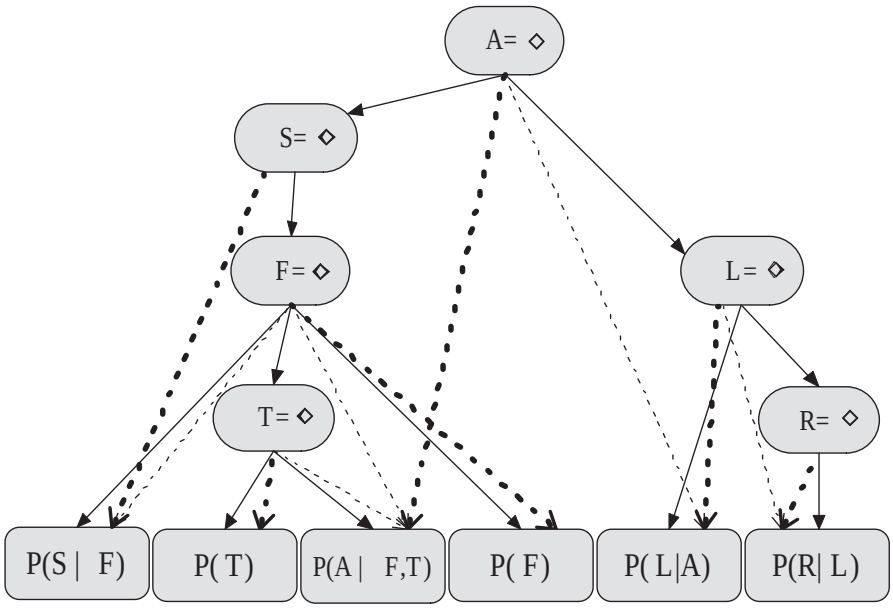


Figure 5.8: The *Fire* conditioning graph, given no evidence. Irrelevant nodes are grayed out.

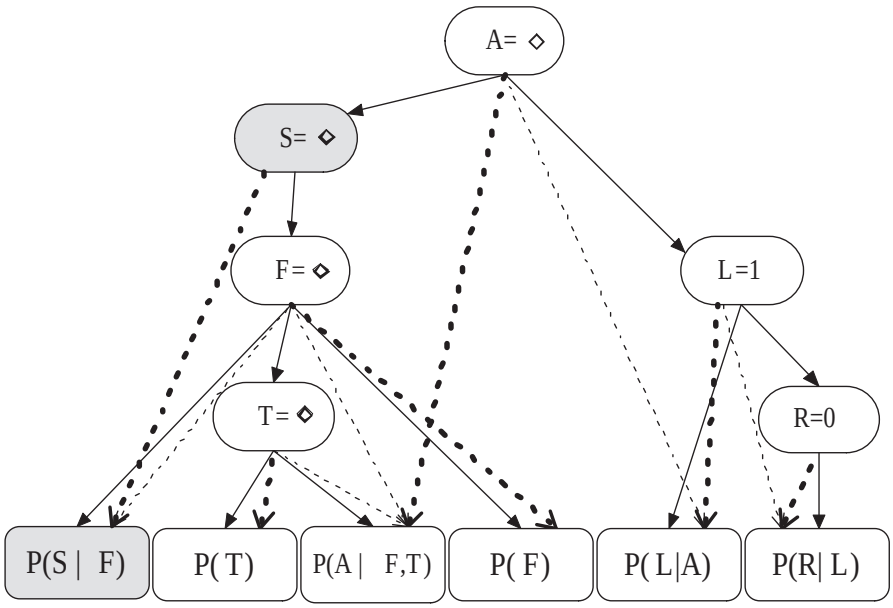


Figure 5.9: The *Fire* conditioning graph, given $L = 1$ and $R = 0$. Irrelevant nodes are grayed out.

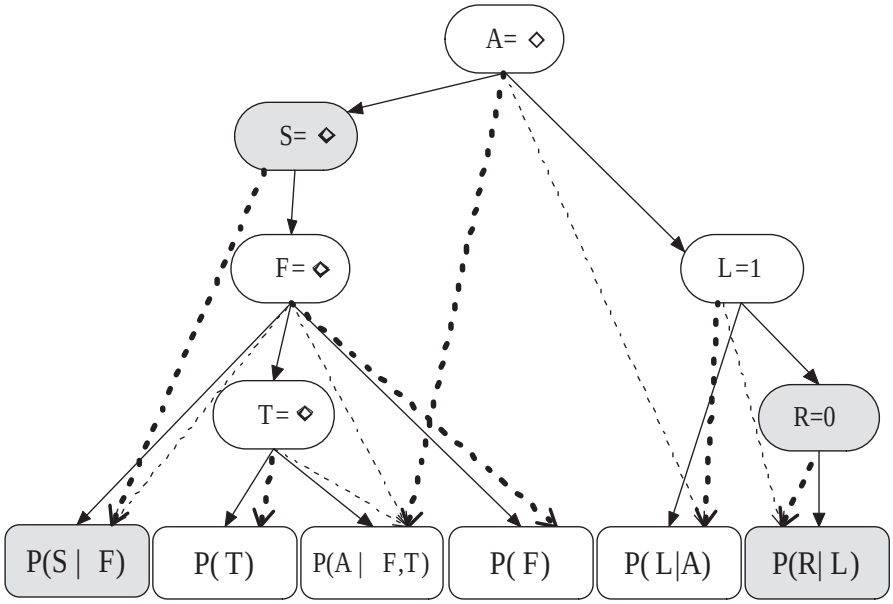


Figure 5.10: The *Fire* conditioning graph, given $L = 1$, $R = 0$, and the query variable F. Irrelevant nodes are grayed out.

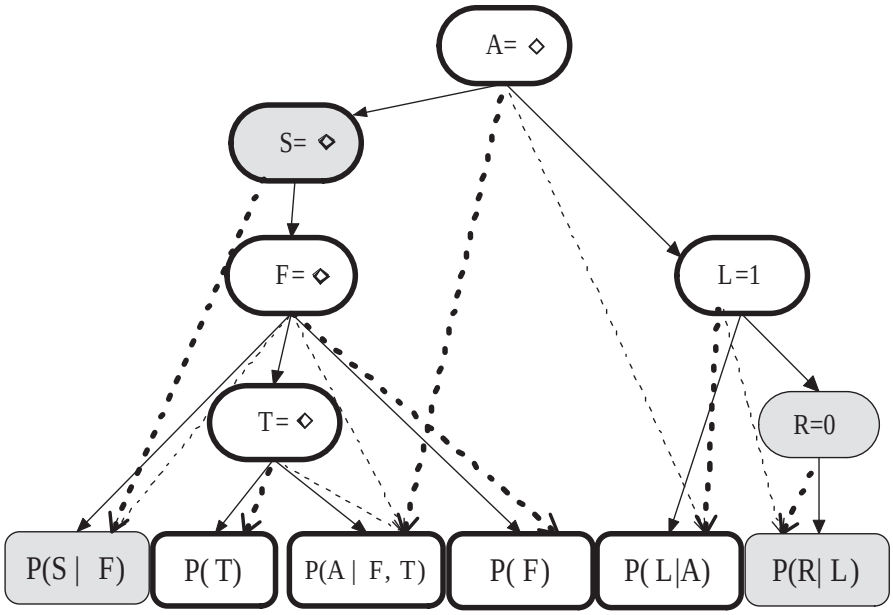


Figure 5.11: The *Fire* conditioning graph, given $L = 1$, $R = 0$, and the query variable F. Irrelevant nodes are grayed out, and active nodes have darkened borders.

```
Query3(N)
 1.  if N is a leaf node
 2.    return N.cpt[N.pos]
 3.  else if N.value ≠ ◇ OR N.relevant = false
 4.    Total ← 1
 5.    for each P' ∈ N.primary s.t. P'.active = true do
 6.      Total ← Total * Query3(P')
 7.    return Total
 8.  else
 9.    Total ← 0
10.    for i ← 0 to N.m − 1 do
11.      SetEvidence2(N, i)
12.      Total ← Total + Query3(N)
13.    SetEvidence2(N, ◇)
14.    return Total
```

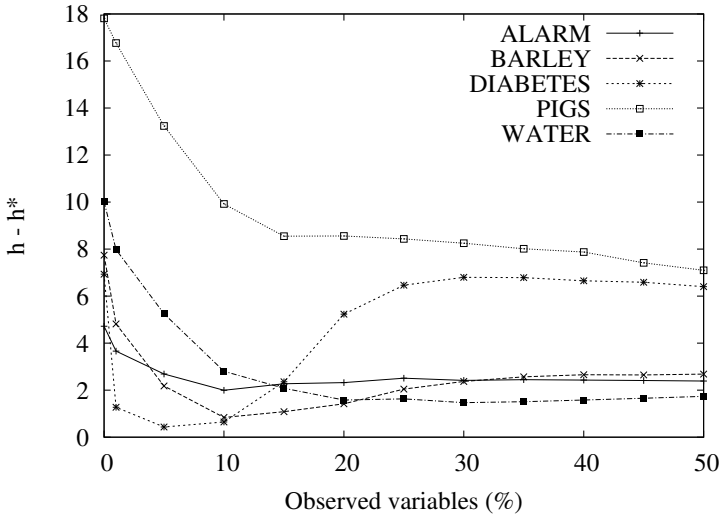
Figure 5.12: The Query algorithm, using active and relevant nodes (Lines 03 and 05).

exponential on its height, which is defined as the maximum number of unobserved, non-query variables along any path from root to leaf (this will be referred to as the graph's *actual height*, and be denoted by h). When barren variables and d-separation are considered, then irrelevant nodes are not conditioned over. The new algorithm (Figure 5.12) is exponential on the maximum number of unobserved, non-query, *relevant* variables along any path from root to leaf. We will refer to this as the conditioning graph's *effective height*, denoted by h^*. Computing using the algorithm in Figure 5.12 will be referred to as computing over the *effective conditioning graph*.

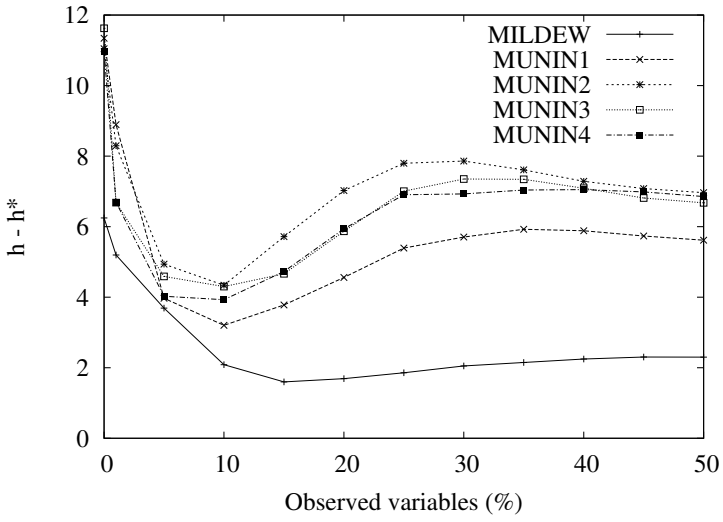
We compared the approaches over the ten repository networks used in Chapter 4 to test our elimination tree construction algorithms. We tested the algorithms using different percentages of evidence variables (ranging from $0 - 50\%$ of the variables in the network). For each test, we generated 100 random sets of evidence, and tested 50 different query variables on for each set of evidence, for a total of 5000 runs *per evidence set size*, per network.

Figure 5.13 shows the difference $h - h^*$ for each network (for readability, we have presented the results in two graphs). The graphs show that ignoring the irrelevant information of the network offers a substantial speedup. The speedup is most prominent when there is no evidence; there is also a tendency for the difference to increase when the percentage of observed variables is greater than 20%. The shapes of the graphs are easily explained by considering where the hardest inference problems are in terms of amount of evidence. When a network has no evidence, the number of barren variables is typically high, so the effective height is low. As evidence is added, the number of barren variables declines, increasing the effective height. However, this increase in the number of variables is eventually offset by the number of d-separated variables, so h^* begins to decline. Hence, the hardest problems for inference in our example networks occur when the amount of evidence is greater than 0% and less than 20%.

To draw a comparison between conditioning graph methods and elimination methods, we compare the effective height of the conditioning graph to the induced width of the elimination variable ordering generated using the min-fill heuristic [38]

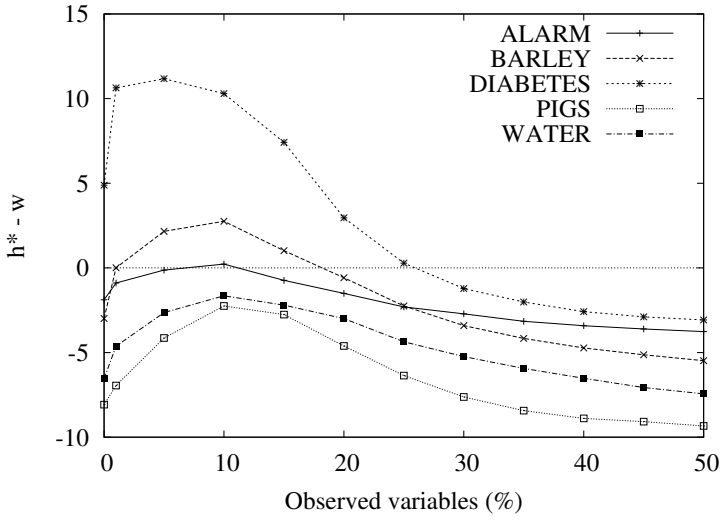


(a)

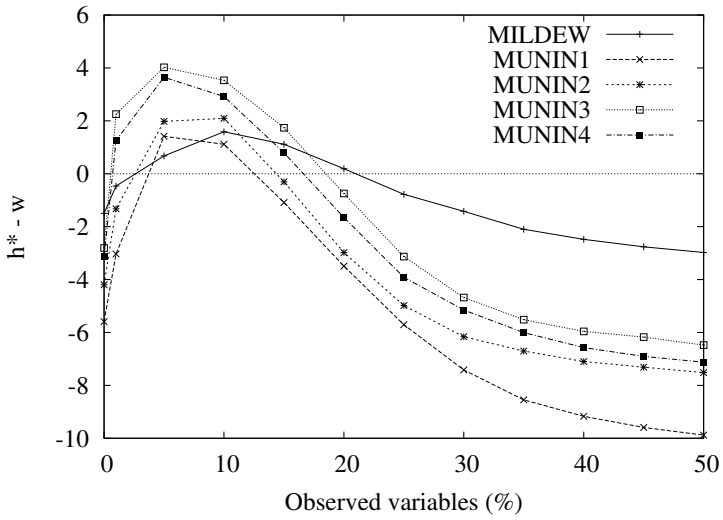


(b)

Figure 5.13: Height difference between actual and relevant conditioning graph.



(a)



(b)

Figure 5.14: Difference between relevant height of conditioning graph and network width.

(denoted by w). By comparing the conditioning graph height to w, we are comparing the complexity of inference in conditioning graphs with the complexity of inference in VE and JTP, by looking at the exponent involved in the worst case analysis. Figure 5.14 shows the result of this comparison. While the actual height of the conditioning graph is typically much worse than the width of the network, the effective height of the relevant conditioning graph is not that much worse than the network width - in fact, it's typically *better* when the amount of evidence is greater than 20%. Note that by better, we mean only that the effective height of the conditioning graph is lower than the width of the network under a consistent variable ordering. This does not mean that the conditioning methods are asymptotically faster than elimination methods (recall that this is not possible), as elimination methods can take advantage of irrelevant information as well. The curves are similar for all graphs: an initial growth, followed by a decline. This shows that in many cases, the complexity of recursive decompositions is within the width of the network, meaning that we obtain reasonable time while not sacrificing the modest memory requirements of the algorithm.

The trends of these graphs are similar for those in the first experiments: the hardest problems occur when the number of observed variables is between 5% and 20% of the total number of variables. The explanation is identical: few observed variables results in many barren variables; many observed variables d-separates much of the network from the query.

5.4 Summary

This section presented application-specific optimizations to the conditioning graph. By ignoring irrelevant information on a per-query basis, we are able to decrease inference runtimes by orders of magnitude. Such optimizations are necessary when computing probabilities using conditioning graphs, as the runtimes are typically much worse than for JTP and VE.

Two classes of optimizations were considered. The first, compile-time optimiza-

tions, were changes made during the construction phase of the conditioning graph. These included removing the sensor variables from the elimination tree, and performing partial elimination (multiplication and marginalization) on subtrees containing no query or evidence variables. Compile-time optimizations have relatively little overhead, as offline computation is typically ignored. However, it is less flexible than runtime optimization, as the evidence and query variables must be known in advance.

We also considered runtime optimizations to the conditioning graph. These were optimizations applied on a per-query basis while the system is online. We dynamically maintained distinct subtrees through the use of a *hood*, and we modified the structure and the algorithm slightly so that we needed only calculate over the relevant information in the graph.

The results of the optimization were positive. In particular, we showed that the irrelevant pruning model could afford conditioning graphs runtimes whose exponent was within the induced width of the network (the upper bound for JTP and VE runtimes). For all of the example networks except for *Diabetes*, the effective heights of their respective conditioning graphs after pruning irrelevant nodes were within 4 variables of the induced width of a heuristic variable ordering *in the worst case*, and were sometimes lower than the width in many cases.

The optimizations presented in this chapter have unique advantages, and the choice of which optimizations to use will depend on the application. The overhead for maintaining separate sensor variables is virtually zero - no additions to the structure are made. The hood increases the complexity of *SetEvidence* and adds variables to the internal nodes. For marking relevant variables, we made further changes to *SetEvidence*, added more variables to the internal nodes, and require a linear-time search through the model to mark the variables before a query can actually take place. While these optimizations get progressively stronger, they also have more overhead. The model (or combination of models) chosen will depend completely on the application (for example, if your network is sufficiently small, like the *Fire* example, then a pre-query linear search to mark irrelevant nodes might be unnecessary).

Chapter 6

Optimization through Caching

Up to this point, conditioning graphs have been presented as a model requiring $\mathbf{O}(n \exp(f))$ space, where f is the size of the largest family in the Bayesian network. Such a model requires much less memory than JTP or VE, facilitating portability to memory-limited applications. However, we have seen that conditioning graphs typically require more runtime than JTP and VE. This penalty can be orders of magnitude, especially in highly connected Bayesian networks. The two previous chapters have presented optimizations for conditioning graphs; however, these optimizations are not necessarily enough to guarantee comparable runtimes to JTP and VE.

We can improve the running time of conditioning graphs by avoiding duplicate calculations, using the caching techniques for recursive decompositions described in Chapter 2. Caching allows calculated values to be stored at internal nodes, which allows for a constant-time lookup the next time the value is needed, rather than recomputing the value. Caching reduces the complexity of inference in recursive decompositions to be equivalent to that of JTP and VE. However, the memory required to cache all values is also asymptotically equivalent to the memory requirements of the standard algorithms. Hence, caching sacrifices memory for time efficiency.

This chapter examines caching as an optimization for conditioning graphs. In Section 6.1, we show how to apply dtree caching to elimination trees and conditioning graphs. Both full and partial caching methods are considered. We also introduce a new technique for pruning the cache, that reduces the memory requirements of full caching while maintaining the same runtime. Finally, we empirically demonstrate the memory requirements for caching in conditioning graphs.

In Section 6.2, we present methods for caching in the *effective conditioning graphs* of Chapter 5, where irrelevant variables are ignored. We will demonstrate techniques for pruning away irrelevant portions of the caches, and show empirically that conditioning graphs employing full caching have fairly modest space requirements in many cases, especially when compared to JTP and VE.

6.1 Caching

Recall that the runtime and memory requirements of JTP and VE are exponential on the induced width of the variable elimination ordering used in their construction/computation, while the time for calculating a value from a conditioning graph is exponential on the height of the graph's underlying elimination tree. Hence, the extra runtime required by conditioning graphs is exponential on this difference, which in some cases can be substantial. Consider again the Bayesian network presented in Figure 4.5. The elimination ordering $\rho = G, F, E, D, C, B, A$ has an induced width of 2, yet the height of the corresponding elimination tree is 7 (Figure 4.6), which means a substantial difference in the running times of JTP/VE and conditioning graphs. For this example, we were able to reduce the underlying elimination tree by choosing a better ordering; however, its height was still greater than 2.

The runtime of computing over an elimination tree can be reduced through the use of caching. We examined caching for dtrees in Section 2.4.1. The same techniques can be applied to elimination trees, without significant modification. As an example, consider again the tree from Figure 4.6, and suppose we are marginalizing variables A and B, which are binary variables. When $A = 0$ (denoted a_0) and $B = 0$ (denoted b_0), the value returned from node C is as follows:

$$Query(N_C) = \sum_C P(C|b_0) \sum_D P(D|C) \sum_E P(E|D) \sum_F P(F|E) \sum_G P(G|F) \qquad (6.1)$$

Notice that this equation does not depend on the value of A. Hence, when $A = 1$ and $B = 0$, the value returned from node C will be exactly the same as the return

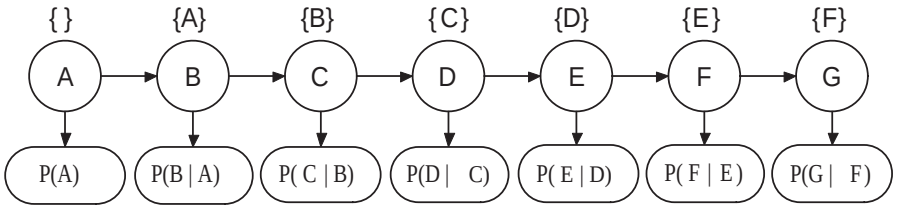


Figure 6.1: Elimination tree of Figure 4.6, with cache-domains shown above each node.

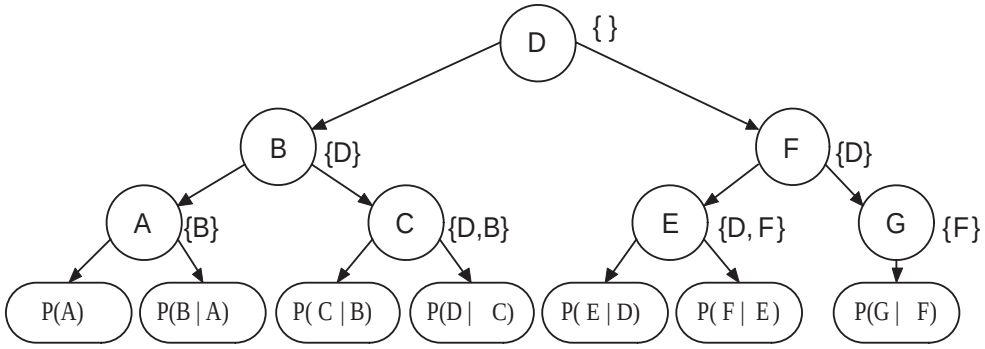


Figure 6.2: Elimination tree of Figure 4.7, with cache-domains shown beside each node.

value of Equation 6.1. By caching this value at node C, it needs only be calculated when $A = 0$, and retrieved when $A = 1$. Similarly, storing the value for when $B = 1$ will produce a similar savings.

The *a-cutset* and *cache-domain* of a node N in an elimination tree are defined to be the same as they were in a dtree; we repeat these definitions here for convenience. The a-cutset of node N is the set of variables labeling the nodes in N's ancestry; the cache-domain of N (denoted $CD(N)$) is the intersection of N's a-cutset and the domains of the CPTs in N's subtree. Figures 6.1 and 6.2 show the cache-domains for the two elimination trees of Figure 4.5. The return value from N depends only on the assignment to its cache-domain, and not its a-cutset. This is clearly demonstrated in the previous example: the cache-domain of node C is $\{B\}$, since the only unique values were for different values of B (that is, not dependent on A).

Changing the algorithm for computing over an elimination tree (Figure 3.5) to allow caching requires very little modification. As with dtrees, when a value is calculated for a particular assignment of the cache-domain, it is stored in the cache

```
P(T, c)
 1.  if T is a leaf node
 2.    return φ_T(c)
 3.  cd ← ⇓_{CD(N)} c
 4.  if cache_T[cd] has been filled
 5.    return cache_T[cd]
 6.  else
 7.    if X_T is instantiated in c
 8.      Total ← 1
 9.      for each T' ∈ ch_T while Total > 0
10.        Total ← Total * P(T', c)
11.    else
12.      Total ← 0
13.      for each x_T ∈ dom(X_T)
14.        Total ← Total + P(T, c ∪ {x_T})
15.    cache_T[cd] ← Total
16.    return Total
```

Figure 6.3: Algorithm for processing an elimination tree given a context.

at that node. When a node is visited, we check to see if the corresponding value is cached. If it is, the cached value is returned; if not, the value is calculated, cached, and returned. Figure 6.3 gives the algorithm P for calculating probabilities from an elimination tree (Figure 3.5), modified to perform caching. We will define the projection of a context $\mathbf{c} \in \mathcal{D}(\mathbf{C})$ to the variables in $\mathbf{D}$, denoted $\Downarrow_{\mathbf{D}} \mathbf{c}$ as the context $\mathbf{d}$ such that: $X \in D$ and $(X = x) \in \mathbf{c} \Rightarrow (X = x) \in \mathbf{d}$. The cache values for node T are stored in an associative array called $cache_T$, which maps the context of T's cache-domain to its corresponding cache entry. The changes required to the algorithm are the addition of Lines 3, 4, and 14.

We now state the time and space complexity of computing over elimination trees

with cache:

Theorem 6.1.1. *The cache space required by an elimination tree is* $\boldsymbol{O}(n \exp(w))$*, where* w *is the width of the variable ordering used to construct the elimination tree. The time required for algorithm* $\mathcal{P}$ *to compute a value from an elimination tree that caches is* $\boldsymbol{O}(n \exp(w))$*.*

Proof. To prove the space complexity bound, note that a cache is simply a distribution over the cache-domain variables. The cache-domain of N represents the variables in the CPTs of N's subtree that do not have a corresponding internal node in N's subtree. Using our variable elimination analogy from Chapter 3, the cache-domain represents variables in the CPTs of N's subtree that have not yet been marginalized out. This means that these cache-domains correspond exactly to the domains of the distributions that would be produced by the VE algorithm using the same variable ordering used to construct the elimination tree. Since the space complexity of elimination is exponential on the width of the variable ordering, the statement of complexity follows.

To prove the time complexity, note that a node in the elimination tree, under caching, will only call its children once for each assignment of its cache-domain and labeling variable. Hence, the total number of recursive calls a node T receives during computation is $\mathbf{O}(\exp(|CD(T_P)| + 1))$, where T_P is the parent node of N. From the VE analogy, $|CD(N)| \leq w$ for all T, hence the complexity follows. □

The time savings afforded by caching can be substantial. Consider the elimination tree shown in Figure 4.6. Without caching, the number of recursive calls to a node T is worst-case exponential on the size of T's a-cutset (Figure 6.4), giving 381 total recursive calls. On the other hand, under full caching, the same graph requires only 49 recursive calls (Figure 6.5).

The graph in Figure 6.4 is somewhat extreme in its height-width ratio. Table 6.1 shows the ratios of height to width in some real-world networks (the same as those used in Chapters 4 and 5). For the height of the elimination trees, we use the average values generated by the heuristics in Chapter 4. For the width of the

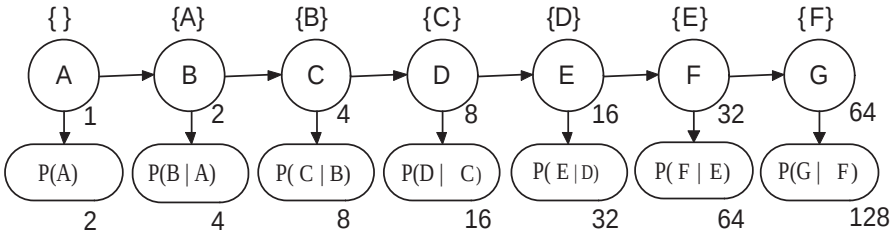


Figure 6.4: Elimination tree of Figure 4.6, with recursive calls shown below each node. Model assumes no caching.

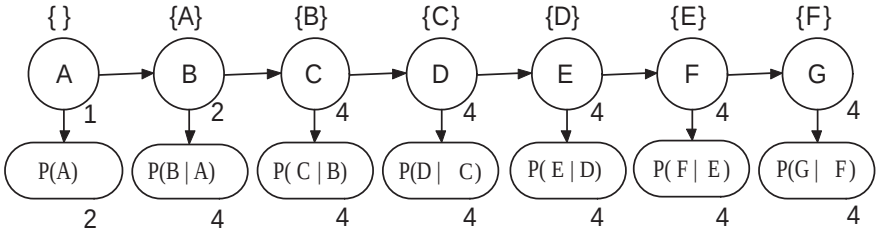


Figure 6.5: Elimination tree of Figure 4.6, with recursive calls shown below each node. Model assumes full caching.

variable ordering, we use the *min-fill* heuristic. The table shows that the differences in height and width can be substantial (9+ variables in the majority of the tested networks).

The addition of caches to elimination trees reduces their runtime complexity to that of JTP and VE. The next section demonstrates how to incorporate caches into conditioning graphs, such that we can achieve these same advantages while maintaining the simplicity and portability of the original conditioning graph model.

6.1.1 Incorporating Caching into Conditioning Graphs

Caching violates our original goals in some ways, as it requires much more space than the original model presented in Chapter 3. Hence, we would like to maintain all other goals when incorporating caching into the conditioning graph model. The model should still be low-level, accessible by non-experts, with a trivial memory footprint and easily assessible.

Fortunately, caching fits very easily into the conditioning graph model. Because the cache is simply a distribution, we can represent and index the cache in the same

Network	Height	Width
Barley	13	7
Diabetes	18	5
Link	37	19
Mildew	9	4
Munin1	18	11
Munin2	16	7
Munin3	16	7
Munin4	17	8
Pigs	19	10
Water	15	11

Table 6.1: Height vs. width of elimination trees on Bayesian networks from the network repository.

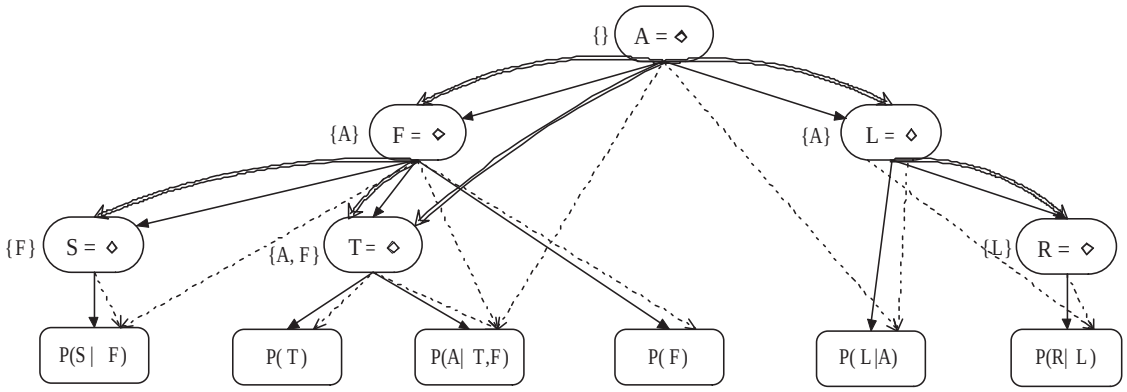


Figure 6.6: The *Fire* conditioning graph, with tertiary arcs (double arcs) added for caching. Cache-domains are shown to the left of each internal node

manner as with a CPT. In fact, we can overload the *cpt* variable at the node to mean a CPT at a leaf node, and cache at an internal node. However, this means that all nodes will now require a *cpt* and *pos* variable, rather than just leaf nodes.

To index into the cache, we need to adjust its *pos* value accordingly. We can accomplish this using the same secondary pointer mechanism that we used to index the CPTs. Hence, to each internal node N, we will add a set of *tertiary* pointers, such that *there is a tertiary arc from an internal node A to an internal node B iff the variable X labeling A is contained in the cache-domain of B*. Figure 6.6 shows the *Fire* conditioning graph, with the tertiary pointers added (cache-domains are shown to the left of each internal node).

The secondary and tertiary arcs are functionally equivalent, indexing the distributions as we traverse the graph. This means that the code for indexing CPTs can be reused, and any optimizations applied to the secondary arcs (such as the scalar values of Section 4.3) can also be applied to tertiary arcs. Given the similarity of secondary and tertiary arcs, it may not be obvious why we need to differentiate between them. However, as we make the model more dynamic (Chapter 5), the sets will need to be handled separately; we do this now for continuity in the algorithms.

When visiting a node during the *Query* algorithm, we need a way to determine whether or not the value to be calculated has already been cached (Line 03 in Figure 6.3). One could imagine a special value that indicates a null entry at the corresponding cache entry, just as $\diamond$ indicates that a variable has no value. However, as a consequence of the way CPTs are indexed in a conditioning graph, it can be shown that for a leaf node N, $N.cpt[i]$ will be calculated before $N.cpt[j]$ for all $j > i$. Therefore, we will use a second integer value, *valid*, that maintains the largest index that contains a valid cache value so far (initially, $N.valid = -1$, for all N). Therefore, when an internal node N is visited, the return value is cached if $N.pos <= N.valid$, and must be calculated otherwise. While requiring extra space, the integer approach has two advantages:

1. Resetting the cache using the integer approach is linear in the number of variables in the graph, while resetting each individual cache value is linear in the size of the caches (which can be exponential on the number of variables).

2. The integer method lends itself well to partial caching, which will be considered in subsequent sections.

Figure 6.7 and 6.8 show the new *SetEvidence* and *Query* algorithms, respectively. Note that we extend the algorithms that employ secondary scalar values (Figures 4.9 and 4.10); the caching scheme would also work for the original algorithm (Figures 3.8 and 3.9). The changes to the non-caching version of the algorithm are minor. In *SetEvidence3*, we simply iterate over the tertiary pointers along with the secondary pointers. In *Query3*, we add one disjunct to the clause of the if statement (Line 1),

SetEvidence3(N, i)

1. $\mathit{diff} \leftarrow i - N.\mathit{value}$ $\{\diamond = 0$ in this equation$\}$
2. **for each** $S' \in N.\mathit{secondary} \cup N.\mathit{tertiary}$ **do**
3. $\quad S'.\mathit{pos} \leftarrow S'.\mathit{pos} + \mathit{scalar}(N, S') * \mathit{diff}$
4. $N.\mathit{value} \leftarrow i$

Figure 6.7: Algorithm for setting evidence, given that we are caching, and secondary scalar values are used.

and we add one line of code to set the value of our cache once it's calculated (Line 14). Hence, while the size of the model is increased by caching, the small memory footprint and simplicity of the function library are maintained.

6.1.2 Partial Caching

The difference in memory between the caching and non-caching conditioning graph can be substantial. An application may have more than enough memory to store and compute over the non-caching model, but not enough to store all of the possible cache values. In such a circumstance, the methods of partial caching should be applied, allowing us to minimize our runtime given a particular memory size. In Section 2.4.1, we examined two different approaches for partial caching. We revisit each of these approaches, and apply them to conditioning graphs.

Non-discrete Uniform Caching

Recall that non-discrete uniform caching allows each node to cache a certain subset of its values, and each node gets to cache exactly the same fraction of its cacheable values. If the current context of the cache-domain corresponds to a value that can be cached, the algorithm executes as though it is caching; if the current context of the cache-domain is not a cacheable value, the algorithm executes as though no caching is taking place.

Non-discrete uniform caching is easily achievable in conditioning graphs. However, rather than caching purely random values, the algorithm caches the first x

```
Query3(N)
 1.  if N is a leaf node OR N.pos ≤ N.valid
 2.    return N.cpt[N.pos]
 3.  else if N.value ≠ ◇
 4.    Total ← 1
 5.    for each P′ ∈ N.primary do
 6.      Total ← Total * Query3(P′)
 7.  else
 8.    Total ← 0
 9.    for i ← 0 to N.m − 1 do
10.      SetEvidence3(N, i)
11.      Total ← Total + Query3(N)
12.    SetEvidence3(N, ◇)
13.  N.cpt[N.pos] ← Total
14.  N.valid ← N.pos
15.  return Total
```

Figure 6.8: Algorithm for querying, given that we are caching, and secondary scalar values are used. Note that cache values must be reset appropriately before calling this algorithm.

values according to the ordering of the contexts in the cache-domain. This allows partial caching to be achieved with very little extension to the algorithm. Define an integer variable called *cache-max* for each internal node, that represents the maximum number of cacheable values at that particular node. The value of *cache-max* will be set when the node is allocated its cache memory. Given this value, Figure 6.9 shows the new version of *Query*. The algorithm itself requires the insertion of one line (Line 14), an if statement that asserts that we only cache (and subsequently mark that memory as cached) if the current index of the context is within the acceptable bound (less than *cache-max*).

```
Query3a(N)
 1.  if N is a leaf node OR N.pos ≤ N.valid
 2.    return N.cpt[N.pos]
 3.  else if N.value ≠ ⋄
 4.    Total ← 1
 5.    for each P′ ∈ N.primary do
 6.      Total ← Total * Query3a(P′)
 7.  else
 8.    Total ← 0
 9.    for i ← 0 to N.m − 1 do
10.      SetEvidence3(N, i)
11.      Total ← Total + Query3a(N)
12.    SetEvidence3(N, ⋄)
13.  if N.pos < N.cache-max
14.    N.cpt[N.pos] ← Total
15.    N.valid ← N.pos
16.  return Total
```

Figure 6.9: Algorithm for querying, given that we are caching, and secondary scalar values are used. Note that cache values must be reset appropriately before calling this algorithm.

Discrete, nonuniform caching

Recall that in a discrete nonuniform caching model, a subset of the nodes are allowed to cache all of their values, while the remainder of the nodes cannot cache any values. The *Cache Allocation Problem* is determining which subset of nodes to cache at in a dtree in order to obtain the fastest computation. As mentioned, there are several algorithms for solving this problem, both optimally [4], and approximately [5]. These algorithms for solving the Cache Allocation Problem in dtrees also work for elimination trees, with minimal modification.

In terms of implementing a discrete, non-uniform partial caching scheme in a

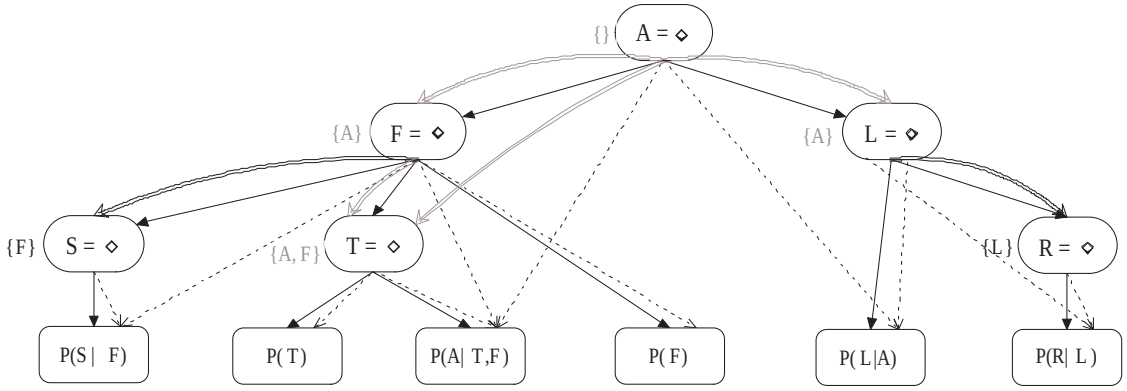


Figure 6.10: The *Fire* conditioning graph, with the dead caches (and corresponding tertiary arcs) grayed out.

conditioning graph, the algorithm in Figure 6.9 is sufficient. Nodes that are to be cached at simply set their *cache-max* values to a value greater than or equal to the maximum number of values they cache, while nodes that do not cache set this value to 0.

6.1.3 Dead Caches

Dead caches are caches whose values are only generated and never queried [3]. Dead caches in dtrees were discussed in Section 2.4.1; dead caches occur in elimination trees and conditioning graphs as well. Consider the *Fire* conditioning graph in Figure 6.6, in particular the *Tampering* (T) node. The cache-domain at this node is $\{A, F\}$. The node is visited only once for each assignment of its cache-domain, therefore, the cache values are never actually used. Dead caches can be removed from recursive decompositions with no runtime consequence. The savings afforded by dead cache removal can be substantial. Figure 6.10 shows the *Fire* example with dead caches removed (grayed out). The live caches require less than a third of the original space required by complete caching.

Dead caches can be identified in dtrees as a cache whose context is a superset of its parent's context. While this definition suffices in elimination trees and conditioning graphs, the restriction of one variable per node allows us to identify dead caches without performing set comparison. Let $var(N)$ represent the variable labeling node

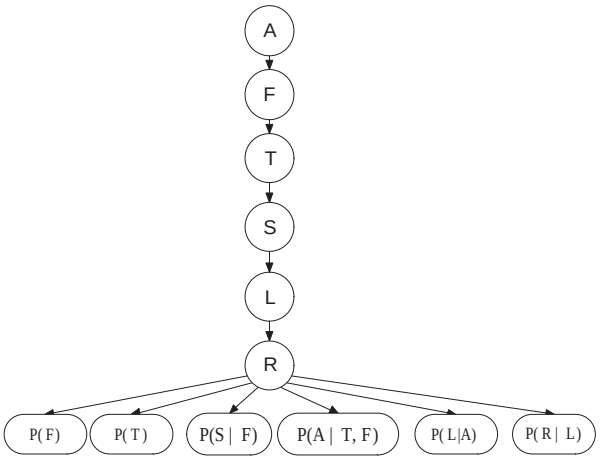


Figure 6.11: The *Fire* elimination tree, in non-proper format.

N, and recall that N_P refers to node N's parent in its elimination tree. We define a *proper conditioning graph* as follows:

Definition 6.1.1. *A conditioning graph (elimination tree) is* ***proper*** *if, for all nodes N, the subtree of node N contains a CPT with $var(N_P)$ in its domain, for all N.*

While this definition seems obvious, it is not necessary for correctness; the algorithms for computing over elimination trees and conditioning graphs will still calculate the correct value when these structures are non-proper, as long as the variables of every CPT occur in the ancestry of that CPT's node. Figure 6.11 shows the *Fire* network, arranged as a non-proper elimination tree. A proper conditioning graph (elimination tree) is a reasonable assumption, and is guaranteed using the construction methods of Chapters 3 and 4. Hence, for the remainder of the chapter, we will assume that our elimination trees and conditioning graphs are proper, unless otherwise stated.

Lemma 6.1.1. *Given node N and its parent node N_P in a proper conditioning graph, $var(N_P) \in CD(N)$.*

Proof. Suppose $CD(N)$ does not include $var(N_P)$. By the definition of a node's cache-domain, this means that N does not contain $var(N_P)$ in its subtree. This violates the definition of a proper conditioning graph. Hence, by contradiction, we can assume that $var(N_P) \in CD(N)$. □

Lemma 6.1.2. *Given a node N and its parent N_P in a proper conditioning graph, $CD(N) - \{var(N_P)\} \subseteq CD(N_P)$.*

Proof. Follows from Lemma 6.1.1 and the fact that all variables in the subtree of N are also in the subtree of N_P. □

We can now prove the following theorem.

Theorem 6.1.2. *If $CD(N) \supset CD(N_P)$, then $CD(N) = CD(N_P) \cup \{var(N_P)\}$.*

Proof. Follows directly from Lemma 6.1.2. □

From Theorem 6.1.2, we can define a dead cache in a conditioning graph to be *a cache whose size is larger than the cache of its parent node.* This is immediate from the example in Figure 6.10 (define the size of the root's cache to be 0). Such a definition allows us to identify dead caches without doing set comparison, and will be extremely useful when we consider dynamic caches in the next section.

The pruning of dead caches (and their corresponding tertiary pointers) can be done as a compile-time step. Computing over a conditioning graph with dead caches can be accomplished using an algorithm such as the one in Figure 6.9, where nodes with dead cache set their *cache-max* values to be 0. In Section 6.2, we will consider caching in effective conditioning graphs developed in Chapter 5. In this case, dead caches occur dynamically, and an algorithm will be required to identify caches on a per-query basis. In this case, identifying dead caches though set sizes, rather than set comparison, allows such an algorithm to be very simple and efficient.

To demonstrate the savings offered by caching in conditioning graphs, Table 6.2 shows the result of removing dead caches from conditioning graphs generated from networks in the Bayesian network repository (the same networks used for testing in Chapter 4). For the remainder of the document, we will refer to computing with dead caches removed as *live caching* (not removing dead caches will be referred to as *complete caching*). The ratios are similar to those obtained by Allen and Darwiche [3]; their results and ours show a substantial reduction in the amount of memory required for caching when dead caching was removed. These results are also useful for comparison to the cache pruning techniques given in the next section.

Table 6.2: The amount of memory required for caching over networks from the Bayesian network repository.

Network	Complete Caching (MB)	Live Caching (MB)
Barley	16.14	7.737
Diabetes	4.172	2.252
Link	1475	16.40
Mildew	1.497	0.4219
Munin1	170.7	90.06
Munin2	3.494	2.065
Munin3	3.605	1.879
Munin4	19.17	6.793
Pigs	1.052	0.4860
Water	10.32	2.170

6.1.4 Subcaching

While live caching requires much less space than complete caching, we can improve on the space requirements further, by noting that while a cache may not be dead, there exist cases where only certain parts of it are live at any moment. Consider a portion of an elimination tree, shown in Figure 6.12. The caches are shown to the left of the node, with dead caches grayed out. There is one live cache at node D, caching values over the variables $\{A, C\}$.

A trace of the visits to node D in Figure 6.12 is given in Table 6.3. While we can see that every entry of the cache is both calculated and reused (no dead entries), there are two points to notice:

1. The cache values corresponding to $A = 0$ are never reused after A becomes 1 (that is, after visit 4).

2. The cache values corresponding to $A = 1$ are only calculated after A becomes 1.

In other words, the portion of the cache corresponding to $A = 0$ is dead following A's being set to 1, so its memory can be reused. As well, the portion of the cache corresponding to $A = 1$ has yet to be calculated following A's conditioning to 1.

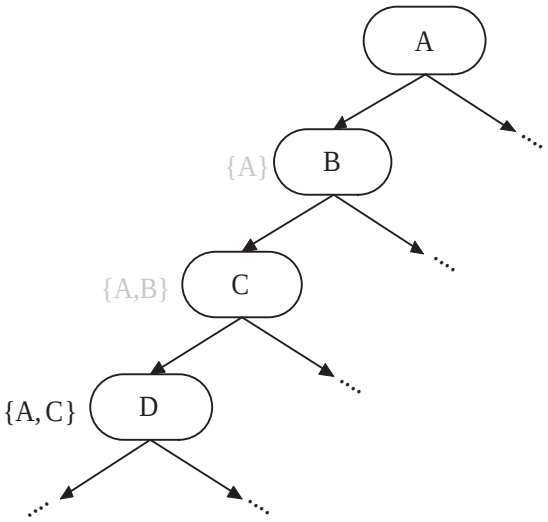


Figure 6.12: A partial elimination tree, with caches shown to the left of the nodes. Dead caches have been grayed out.

Table 6.3: Trace of visits to node D in Figure 6.12.

Visit	**Context**	**Cache-domain**	**Cached?**	**Comment**
1	$A = 0, B = 0, C = 0$	$A = 0, C = 0$	No	
2	$A = 0, B = 0, C = 1$	$A = 0, C = 1$	No	
3	$A = 0, B = 1, C = 0$	$A = 0, C = 0$	Yes	Cached at visit 1
4	$A = 0, B = 1, C = 1$	$A = 0, C = 1$	Yes	Cached at visit 2
5	$A = 1, B = 0, C = 0$	$A = 1, C = 0$	No	
6	$A = 1, B = 0, C = 1$	$A = 1, C = 1$	No	
7	$A = 1, B = 1, C = 0$	$A = 1, C = 0$	Yes	Cached at visit 5
8	$A = 1, B = 1, C = 1$	$A = 1, C = 1$	Yes	Cached at visit 6

Table 6.4: Trace of visits to node D in Figure 6.12.

Visit	Context	Cache-domain	Cached?	Comment
1	$A=0, B=0, C=0$	$C=0$	No	
2	$A=0, B=0, C=1$	$C=1$	No	
3	$A=0, B=1, C=0$	$C=0$	Yes	Cached at visit 1
4	$A=0, B=1, C=1$	$C=1$	Yes	Cached at visit 2
Cache is reset!!				
5	$A=1, B=0, C=0$	$C=0$	No	
6	$A=1, B=0, C=1$	$C=1$	No	
7	$A=1, B=1, C=0$	$C=0$	Yes	Cached at visit 5
8	$A=1, B=1, C=1$	$C=1$	Yes	Cached at visit 6

Therefore, *these values can occupy the same memory.* The new cache will be indexed only on the variable C, and will be reset each time the value of A changes. Table 6.4 shows the new trace given this system. While the computation at node D has not changed, we have reduced its cache memory by 50%.

This form of caching is not partial caching, as we do eventually cache all of the values. However, since we only cache a subset of the entire cache at a time, we refer to this as *subcaching.* The subset of the cache-domain of node N that will define the cache will be referred to as an *effective cache-domain*, denoted $ECD(N)$.

The effective cache-domain can be defined as follows: let $\rho = [A_1, ..., A_q]$ be the cache-domain of node N ordered according to the elimination tree (in the same way we order variables in CPTs). Let $\rho' = [B_1, ..., B_r, var(N_P)]$ be the cache-domain of the parent node of N, ordered according to the elimination tree, and appended with the variable labeling N_P. We will use the notation $\rho[i]$ to denote the ith variable in ρ according to the said ordering. The effective cache-domain of N, denoted $ECD(N)$, is equal to $[A_i, ..., A_q]$, where $\rho[i] \neq \rho'[i]$ and $\forall j < i \;\; \rho[j] = \rho'[j]$. The cache will be reset each time the value of A_{i-1} changes (if it exists). We will denote this variable as the *reset variable* of N.

There are two special cases that the above specification of effective cache-domains does not directly address, and should be clarified:

1. $A_1 \neq B_1$. This means that the cache-domain of N is equivalent to the effective cache domain of N, in which case the cache is never reset. Caching proceeds

as normal.

2. $A_i = B_i, \forall i$. This means that the effective cache-domain of N is empty. However, this also means that $CD(N) = CD(N_P) \cup \{var(N_P)\}$, which we proved previously indicates a dead cache. Hence, an empty effective cache-domain indicates a dead cache.

The following theorem proves the correctness of reusing memory in subcaching:

Theorem 6.1.3. *When the value of N's reset variable changes, no current cache values at N will ever be queried again.*

Proof. Let $\rho = [A_1, ..., A_q]$ be an ordering over the cache-domain of N, and let $\rho' = [B_1, ..., B_r, var(N_P)]$ be an ordering over N_P's cache-domain and N_P's variable, ordered as defined above. Note that ρ' is a superset of ρ. Let A_{i-1} be the reset variable for node N. Let $\mathbf{p'} \in \mathcal{D}(\rho')$ be a context over the variables in ρ'. Similar to our previous definition, we define the projection of a context $\mathbf{p'} \in \mathcal{D}(\rho')$ to the variables in ρ, denoted $\Downarrow_\rho \mathbf{p'}$ as the context $\mathbf{p}$ such that $X \in \rho$ and $(X = x) \in \mathbf{p'} \Rightarrow (X = x) \in \mathbf{p}$.

In order for a cache value to be successfully hit, there must exist two contexts $\mathbf{p'_1}$ and $\mathbf{p'_2} \in \mathcal{D}(\rho')$ such that $\mathbf{p'_1} \neq \mathbf{p'_2}$ and $\Downarrow_\rho \mathbf{p'_1} = \Downarrow_\rho \mathbf{p'_2}$. This means that there exists a variable $Y \in \rho' - \rho$ such that $\Downarrow_{\{Y\}} \mathbf{p'_1} \neq \Downarrow_{\{Y\}} \mathbf{p'_2}$. For two contexts to be the same subsequent to a change in the value of A_{i-1}, one of these variables that have different values in $\mathbf{p'_1}$ and $\mathbf{p'_2}$ must exist in the ancestry of A_{i-1}'s node. However, no variable exists that can meet all these criteria at once: if Y is in $CD(N_P)$ and is before A_{i-1} in the ordering, then it must also exist in $CD(N)$, which contradicts the statement $Y \in \rho' - \rho$. Hence, we conclude that the two contexts $\mathbf{p'_1}$ and $\mathbf{p'_2}$ cannot be generated across the changing of the value of A_{i-1}.

□

As with dead cache removal, computing subcaches and setting up the appropriate tertiary pointers can be accomplished at compile-time. A node N has a tertiary pointer to each node that includes $var(N)$ in its subcache. The only thing remaining

Table 6.5: The amount of memory required for caching over networks from the Bayesian network repository.

Network	Complete Caching (MB)	Live Caching (MB)	Subcaching (MB)
Barley	16.14	7.737	0.4580
Diabetes	4.172	2.252	0.6250
Link	1475	16.40	12.20
Mildew	1.497	0.4219	0.1058
Munin1	170.7	90.06	41.81
Munin2	3.494	2.065	0.4560
Munin3	3.605	1.879	0.6227
Munin4	19.17	6.793	0.4271
Pigs	1.052	0.4860	0.1234
Water	10.32	2.170	0.2068

is the functionality that determines when a cache should be reset. To accomplish this, we add tertiary pointers from a node N to any node whose reset variable is $var(N)$. As mentioned before, the cache at node N is reset when the value of N's reset variable changes.

For simplicity sake, we do not wish to differentiate between standard tertiary pointers, and those that represent arcs from reset variables. We can accomplish this in a simple fashion by noting that the value of N's reset variable can be calculated as $N.pos$ divided by the total size of the cache (denoted as $N.ccsize$). Hence, we can indirectly monitor the value of N's reset variable from node N, and reset the cache whenever that value changes. To accomplish this, we will keep the value $N.pos/N.ccsize$ saved in a variable called $N.reset$. Figure 6.13 shows the *Query* algorithm modified to accommodate subcaches.

Table 6.5 compares the memory requirements of complete caching, live caching, and subcaching over the Bayesian networks we have used for testing. The results empirically demonstrate that subcaching reduces the overall size of the caches considerably, even from live caching. Eight of the ten networks required less than 1 MB of cache storage. This reduction in space does not affect the time complexity of computation over the graph – it remains $\mathbf{O}(n \exp(w))$.

```
Query4(N)
 1.  if N is not a leaf node AND N.reset ≠ N.pos/N.ccsize
 2.    N.reset ← N.pos / N.ccsize
 3.    N.valid ← −1
 4.  if N is a leaf node OR N.pos ≤ N.valid
 5.    return N.cpt[N.pos]
 6.  else if N.value ≠ ⋄
 7.    Total ← 1
 8.    for each P′ ∈ N.primary do
 9.      Total ← Total ∗ Query4(P′)
10.  else
11.    Total ← 0
12.    for i ← 0 to N.m − 1 do
13.      SetEvidence3(N, i)
14.      Total ← Total + Query4(N)
15.    SetEvidence3(N, ⋄)
16.  N.cpt[N.pos] ← Total
17.  N.valid ← N.pos
18.  return Total
```

Figure 6.13: Algorithm for querying, given that we are subcaching.

6.2 Caching at Runtime

The previous section demonstrated methods for caching in a conditioning graph. Caching allows computation over elimination trees and conditioning graphs to be asymptotically as fast as JTP and VE, however, this speedup comes at the expense of space. Removing dead caches and subcaching reduced the memory requirements of caching substantially; we can further improve this result by caching only over variables that directly pertain to the query and current evidence.

In the second part of Chapter 5, we examined runtime optimizations to the conditioning graph model that exploited independencies occurring as the result of specific queries and evidence in the original Bayesian network. In this section, we will revisit this optimization technique, in an effort to reduce the space requirements of caching in conditioning graphs. Since we ignore variables that are irrelevant to the current query and evidence, then these variables will also be excluded from cache. We will demonstrate empirically that the memory requirements of caching can be decreased once these methods are deployed.

To simplify the discussion, no partial caching will be considered in this section; we will assume that sufficient memory exists to cache all the values we need. Extending these methods to partial caching models is left as future work.

The algorithm for dynamically allocating cache is as follows:

1. Identify the potential candidates for the cache-domains of each node.

2. Determine the cache-domains for each node N from the set of potential candidates, which we will refer to as the *relevant cache-domain* (denoted $CD_R(N)$).

3. Allocate sufficient memory for each context.

Step 1 of the above algorithm, identifying potential candidates for a cache, is simple once we have determined the relevant variables for a particular problem. First, irrelevant variables are not needed in a cache-domain, as we do not iterate over the values of an irrelevant variable. Secondly, an observed variable is not needed in a

cache-domain, as the value of the variable is always the same, so the cache value is not dependent on the value of that variable. Hence, the potential candidates for cache-domains are the relevant unobserved variables. These variables can be identified after the *SetRelevant* algorithm is called (Figure 5.7).

Step 3 of the algorithm is straightforward once the relevant cache domains have been established, as the size of each cache is a function of its relevant cache-domain. The allocation of cache memory to each node is implementation-dependent; in a simple system, it may be a matter of setting the cache pointer of each node to a particular memory location; in a high level language like C, a command like *malloc* may be used.

Step 2 of the algorithm, determining the relevant cache-domains for each node, is the most involved step, and is the focus of this section. Recall that the cache-domain of a node is identified as the intersection of the set of variables in that node's ancestry with the set of variables in the CPTs of its leaves. This cache-domain will be a superset of the node's relevant cache-domain. Hence, the relevant cache-domain at each node can be calculated as the intersection of its cache-domain and the set of potential candidates. We can use the tertiary pointers outlined in Section 6.1 to build the relevant cache-domain incrementally. Figure 6.14 shows the algorithm *MakeCache*, that traverses through the conditioning graph and determines the cache size at each node. The size of the cache at node N will be specified as $N.ccsize$, which initially takes the value 1. *MakeCache* traverses the graph depth-first, and at each node containing an unobserved, relevant variable, it multiplies the cache-size of its tertiary children by the size of its variable's domain. Note that the *MakeCache* algorithm is linear on the number of tertiary arcs in the conditioning graph, which is in the worst case quadratic in the number of nodes in the graph.

In the previous section, we defined live-caching, and showed how live caching uses much less memory than complete caching. When irrelevant and observed variables are not ignored, dead caches can be identified at compile-time. However, when considering only relevant cache-domains, a cache can become dead as a result of specific queries and evidence. Recall that a cache at a node N is dead if its cache-

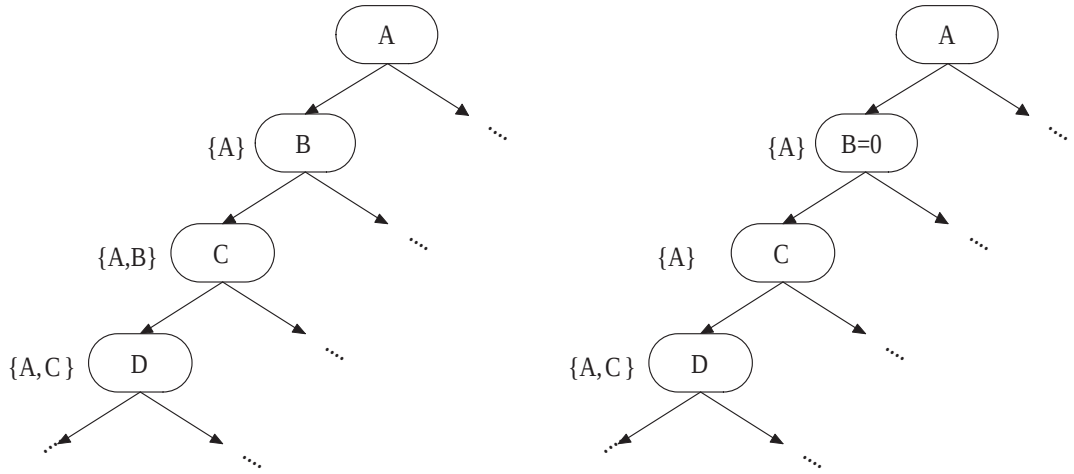
```
MakeCache(N)
1.  if N is a leaf node
2.    return
3.  else if N.value = ◇ AND N.relevant = true
4.    for each N' ∈ N.tertiary s.t. N'.active = true do
5.      N'.ccsize ← N'.ccsize * N.size
6.  for each P' ∈ N.primary s.t. P'.active = true do
7.    MakeCache(P')
```

Figure 6.14: The *MakeCache* algorithm, specifying the size of caches. Note that *MakeCache* must be run after *SetRelevant*.

domain is a superset of the cache-domain at N_P. In other words, if N's cache is not dead, then the cache-domain at N_P contains a variable or set of variables that is not in the cache-domain of N. If these variables were to become observed or irrelevant, then they would be removed from the cache-domain of N_P, and the node's cache becomes dead. Figure 6.15 shows an example of this. The cache at D is not a dead cache – until the variable B is observed.

While a live cache can become dead as a result of observation or irrelevant variables, it should be noted that the reverse is not true. That is, a dead cache cannot become 'live' as a result of a variable being observed or irrelevant. The proof of this statement is simple: suppose the cache-domain at node N is dead. Pruning a variable from the cache-domain of N's parent will still make N's cache-domain a subset of N's cache-domain. And any variable pruned from N's cache-domain will also be pruned from the cache-domain of N's parent. Hence, in both cases, the subset relationship between the cache-domain is maintained.

Because dead caches can occur at runtime, we require an extension to the *MakeCache* algorithm that determines whether or not a cache is dead. We can use the rule of the last section: in a proper conditioning graph, a cache at a node is dead if its cache-domain is larger than the cache-domain of its parent node. While this rule is still correct in the effective conditioning graph, it misses dead caches in nodes



(a) A partial elimination tree, with all variables unobserved. The cache at node D is not dead.

(b) The elimination tree, with node B observed. B is removed from C's cache-domain, and the cache at node D is now dead.

Figure 6.15: An example of a cache becoming 'dead' because of evidence.

whose parents contain an observed or irrelevant variable. Figure 6.15 demonstrates this clearly. The cache at node C is dead, regardless of B being observed. However, when B is observed, the cache at node C is not larger than the cache at node B; they are the same size. Hence, we require a second indicator for dead caches:

Theorem 6.2.1. *If the number of variables in the relevant cache-domain of node N is equal to the number of variables in the relevant cache-domain of N_P and the variable at N_P is observed or irrelevant, then the cache at N is dead.*

Proof. It suffices to prove that the relevant cache-domains of the two nodes are identical. We know from Lemma 6.1.2 that $CD(N) - var(N_P) \subseteq CD(N_P)$. Any irrelevant or observed variable pruned from $CD(N)$ is also pruned from $CD(N_P)$, hence, $CD_R(N) - var(N_P) \subseteq CD_R(N_P)$. Since the variable at N_P is observed or irrelevant, it is not included in $CD_R(N)$, therefore, $CD_R(N) \subseteq CD_R(N_P)$. Since the sizes of $CD_R(N)$ and $CD_R(N_P)$ are the same, it follows that $CD_R(N) = CD_R(N_P)$. □

Given Theorem 6.2.1, we can now determine whether a cache is dead or not, regardless of pruned variables. Figure 6.16 shows the algorithm for determining the

cache sizes at each node, and whether or not the cache is dead. We require two more values at each internal node. Given a node N, the value $N.cc$ is an integer representing the number of variables in that node's relevant cache-domain, which initially takes value 0. $N.dead$ is a boolean value that is true whenever the cache at that node is dead. We have also added a function called *on*, that takes two conditioning graph nodes as parameters. The function $on(N, N')$ will return $true$ if the tertiary arc between N and N' is active, and $false$ otherwise. Recall that a tertiary pointer exists from node A to node B if node B's cache-domain contains the variable at node A. If B's cache is dead, then the tertiary arc from A to B will simply be "turned off". Algorithm 6.17 shows the new *SetEvidence* algorithm for the *Query* algorithm to use, calculating only over tertiary arcs that are active (note that $N.tertiary_{on} = \{n \in N.tertiary | on(N, n) = true\}$).

In the last section, we defined subcaching, which allowed a further reduction in the memory requirements of caching while maintaining the same time complexity. The incremental building of the relevant cache-domains allows us to elegantly find the effective cache-domain of each node. Recall that the subcache of a node can be identified by ordering the variables in the cache-domains according to their depth in the elimination tree, and removing the prefix of each cache that is in common with its parent cache. Because we are building the caches incrementally, it follows that as soon as the parent of node N acquires a variable in its relevant cache-domain that is not in N's relevant cache-domain, all subsequent variables added to N's relevant cache-domain are part of its effective cache-domain. And because we know that all variables in N's relevant cache-domain must exist in N_P's relevant cache-domain (with the exception of $var(N_P)$), we need only check the sizes of the caches as they are constructed. If N_P's cache becomes larger than N's cache, then N accepts all remaining cache-domain variables into its effective cache-domain. This avoids a set comparison of variables between the two nodes.

Figure 6.18 shows the algorithm for dynamically setting the subcaches. The tertiary pointers at each node must be ordered according to the height of their respective nodes in the tree (higher nodes appear earlier in the ordering than lower

```
MakeCache1(N)
 1.  if N is a leaf node
 2.    return
 3.  else if N.value = ◇ AND N.relevant = true
 4.    for each N' ∈ N.tertiary s.t. N'.active = true do
 5.      N'.cc ← N'.cc + 1
 6.  for each P' ∈ N.primary s.t. P'.active = true do
 7.    MakeCache1(P')
 8.  if N.value = ◇ AND N.relevant = true
 9.    for each N' ∈ N.tertiary s.t. N'.active = true do
10.      if N'.dead = true
11.        on(N, N') ← false
12.      else
13.        N'.ccsize ← N'.ccsize * N.size
14.        on(N, N') ← true
15.  if N.parent = null
16.    N.dead = true
17.  else if N.parent.value = ◇ AND N.parent.relevant = true
18.    N.dead ← N.cc > N.parent.cc
19.  else
20.    N.dead ← N.cc = N.parent.cc
```

Figure 6.16: The *MakeCache* algorithm, specifying the size of caches, and labeling dead caches.

SetEvidence4(N, i)

1. $\mathit{diff} \leftarrow i - N.\mathit{value}$ $\{\diamond = 0$ in this equation$\}$
2. **for each** $S' \in N.\mathit{secondary} \cup N.\mathit{tertiary}_{on}$ **do**
3. $\quad S'.\mathit{pos} \leftarrow S'.\mathit{pos} + \mathit{scalar}(N, S') * \mathit{diff}$
4. $N.\mathit{value} \leftarrow i$

Figure 6.17: Algorithm for setting evidence, given that caching and secondary scalar values are used.

nodes). We add the boolean value *sub* to each node. For a node N, $N.\mathit{sub}$ is initially false, and remains false until its parent node acquires a variable in its cache-domain that is not in its own cache-domain. Once this value becomes true, then all subsequent variables that attempt to add themselves to the cache-domain are added.

In the last section, application-specific information, such as relevant variables and observed variables, was not considered when constructing caches. In a proper conditioning graph, this meant that the cache-domain at any non-root node was guaranteed to contain at least one variable (the variable from its parent node). An empty subcache indicated a dead cache. However, when relevant and observed variables are pruned from cache-domains, it can leave empty cache-domains, which also makes empty subcaches. However, these caches should not be considered dead; it just means that the value returned by that node will always be the same, and not depend on the value of any variables in the node's ancestry. Figure 6.19 shows an example of an empty cache-domain occuring. When B is observed, the cache-domain at node A becomes empty. However, this cache is not dead, it will just always return the same value. Hence, we need a way to differentiate empty subcache-domains into dead caches and single-value contexts. The definition of a dead-cache does not change; a cache is dead if its context is a superset of the cache-domain of its parent node. Hence, the code from *MakeCache1* for identifying dead caches (Lines 14-19) is included in *MakeCache2* (Lines 6-11).

```
MakeCache2(N)
 1.  if N is a leaf node
 2.    return
 3.  else if N.value = ◇ AND N.relevant = true
 4.    for each N' ∈ N.tertiary s.t. N'.active = true do
 5.      on(N, N') ← SubCache(N, N.size, N')
 6.  if N.parent = null
 7.    N.dead = true
 8.  else if N.parent.value = ◇ AND N.parent.relevant = true
 9.    N.dead ← N.cc > N.parent.cc
10.  else
11.    N.dead ← N.cc = N.parent.cc
12.  for each P' ∈ N.primary s.t. P'.active = true do
13.    MakeCache2(P')
SubCache(N, i, N')
 1.  N'.cc ← N'.cc + 1
 2.  if N'.sub OR N'.cc < N'.parent.cc OR (N = N'.parent AND N'.cc = N.cc)
 3.    N'.ccsize ← N'.ccsize * i
 4.    N'.sub = true
 5.    return true
 6.  return false
```

Figure 6.18: The *MakeCache* algorithm, specifying the size of *sub-caches*, and labeling dead caches.

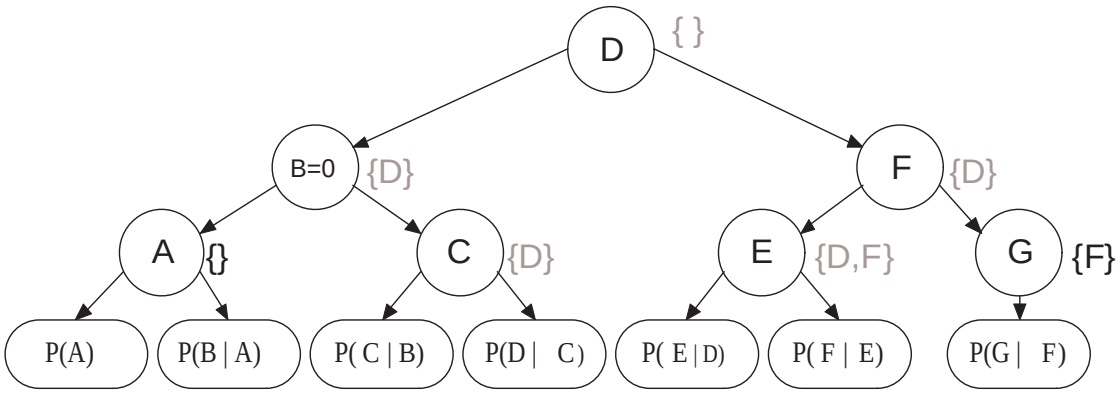


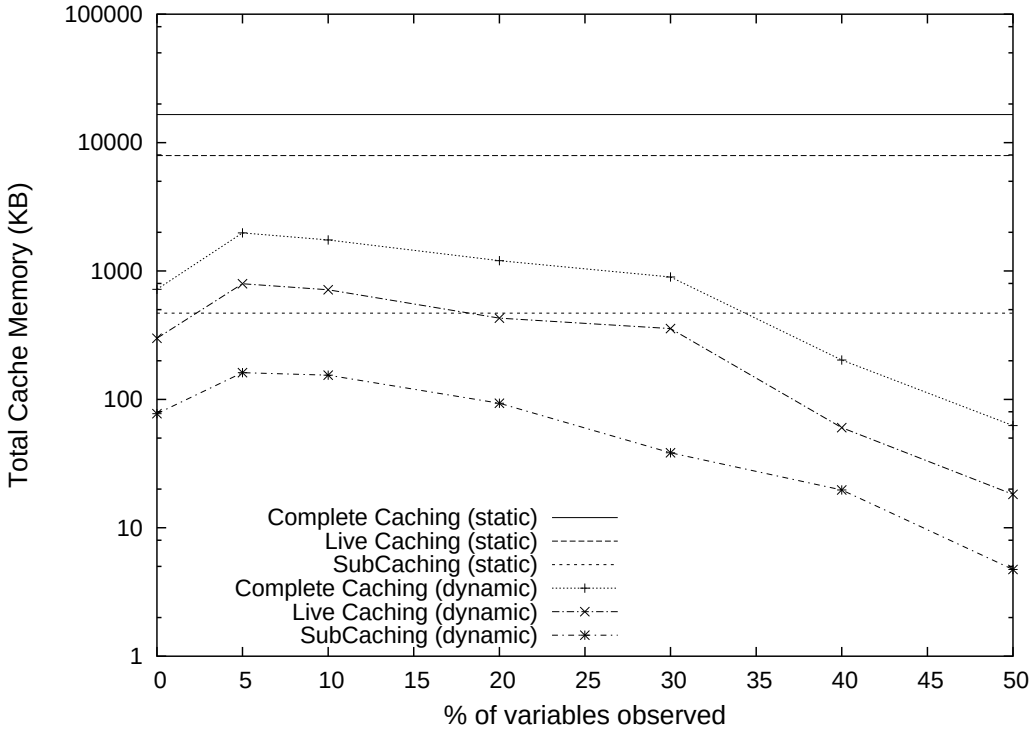
Figure 6.19: An example of an elimination tree, where the evidence creates an empty cache-domain (node A). Dead caches are grayed out.

6.2.1 Evaluation

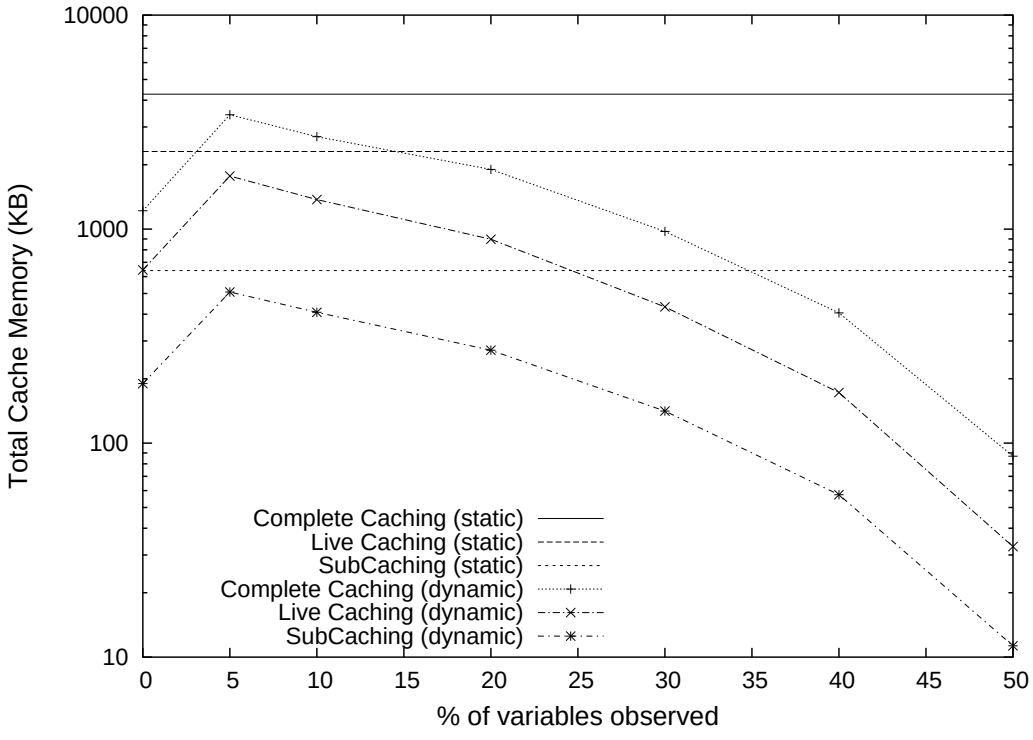
To evaluate the memory requirements of caching in the effective conditioning graph, we tested the above algorithms over the same repository networks used for testing in Chapter 5. For each network, we tested the cache memory requirements over different amounts of memory. For each network/memory configuration, 50 random evidence sets were generated, and the memory requirements were recorded for complete caching, live caching, and subcaching. The average of these values are plotted in Figures 6.20 through 6.24. For comparison purposes, the memory requirements for complete caching, live caching, and subcaching when no graph pruning occurs are also plotted (these are referred to as static, whereas the others are referred to as dynamic).

From the graphs, we can see a substantial decrease in the memory requirements when only relevant information is used. For some networks (Munin1, Pigs), by only considering relevant information, the memory requirements for complete caching in the effective conditioning graph were less than the memory required for subcaching in the actual conditioning graph. We also see the same trend as in Chapter 5: the harder problems (number of observed variables between 5% and 20%) created the largest demands for memory.

One of the strongest points to be taken from these results is that the actual memory requirements to compute probabilities from a Bayesian network in $\mathbf{O}(n \exp(w))$

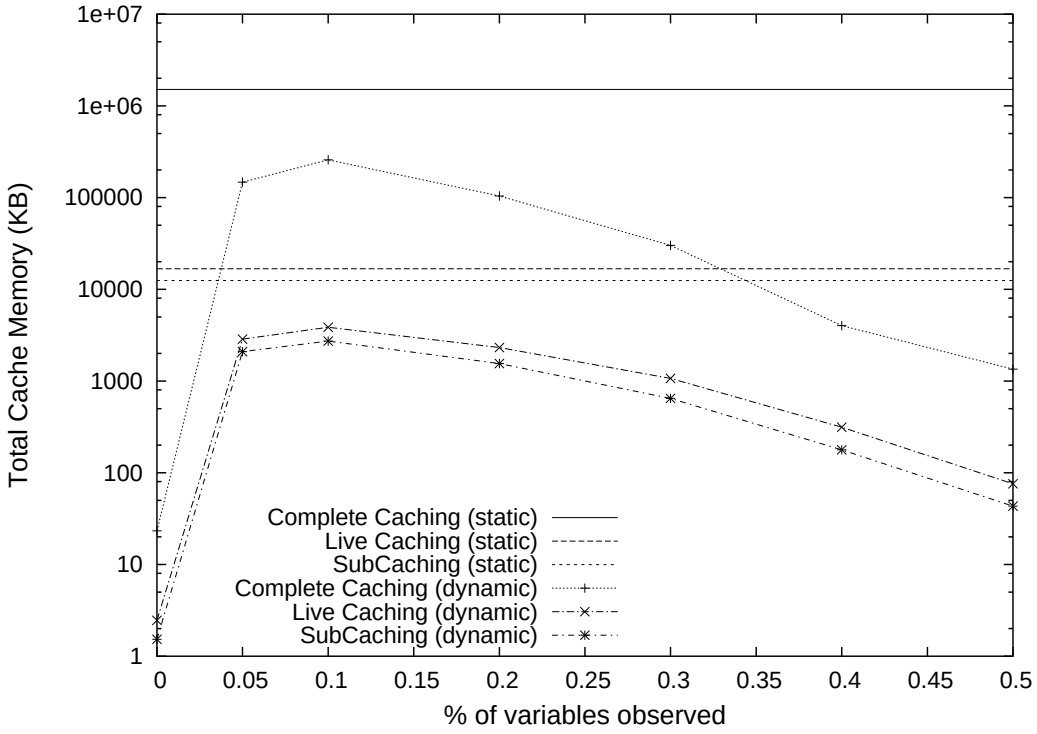


(a) Barley

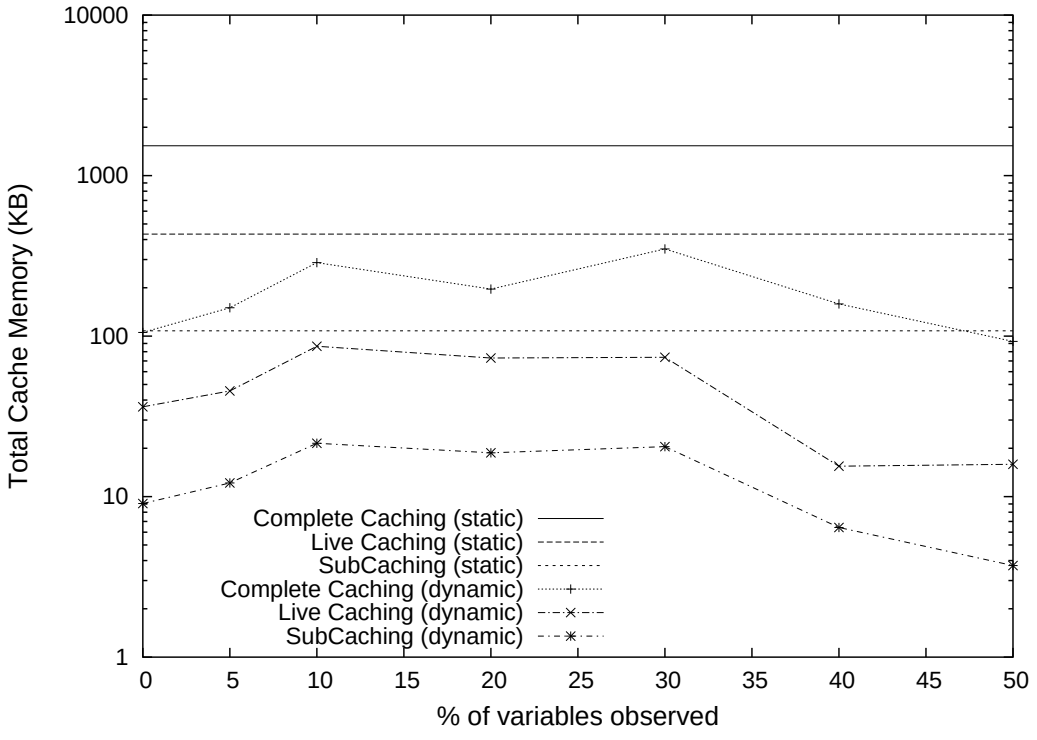


(b) Diabetes

Figure 6.20: Memory requirements for caching in effective conditioning graphs vs. actual conditioning graphs.

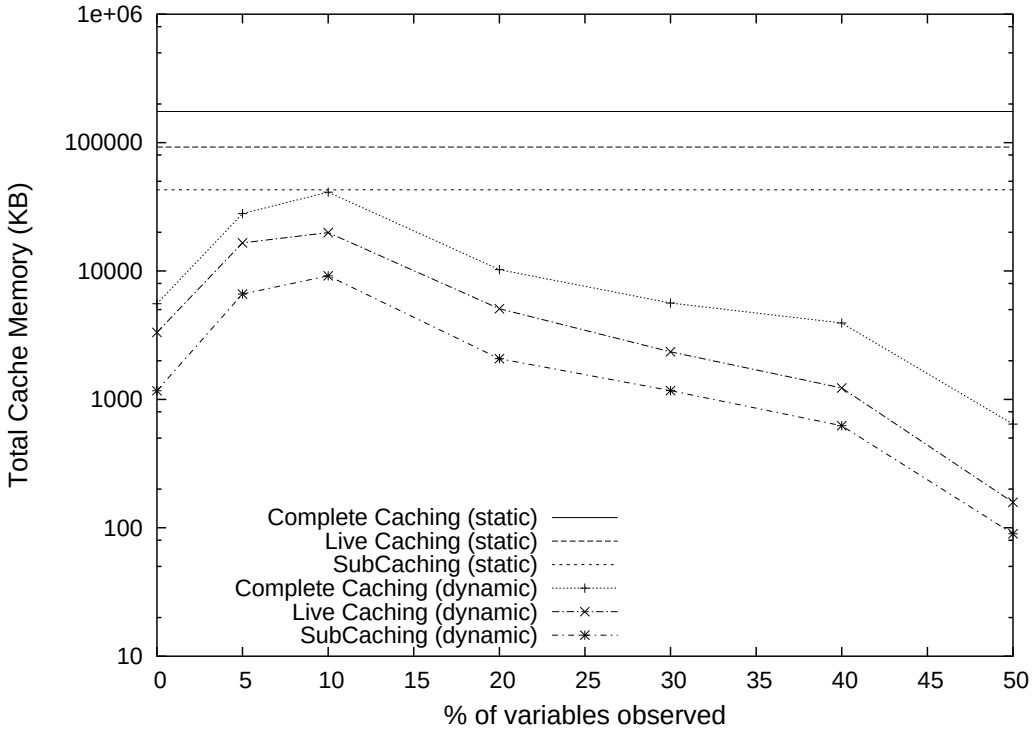


(a) Link

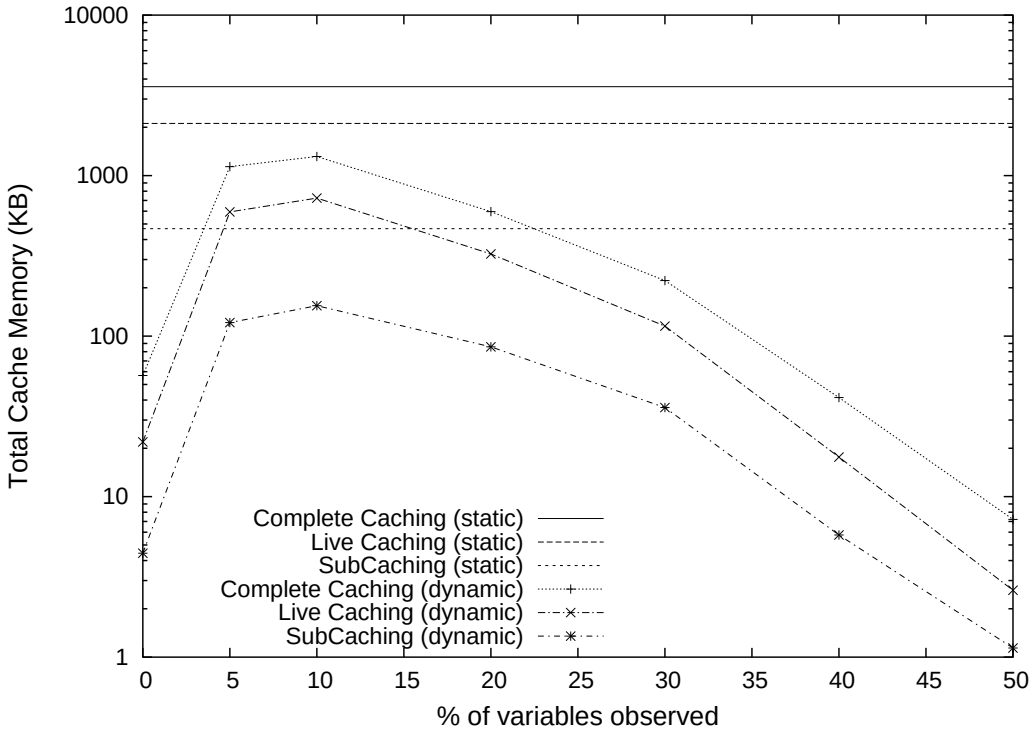


(b) Mildew

Figure 6.21: Memory requirements for caching in effective conditioning graphs vs. actual conditioning graphs (continued).

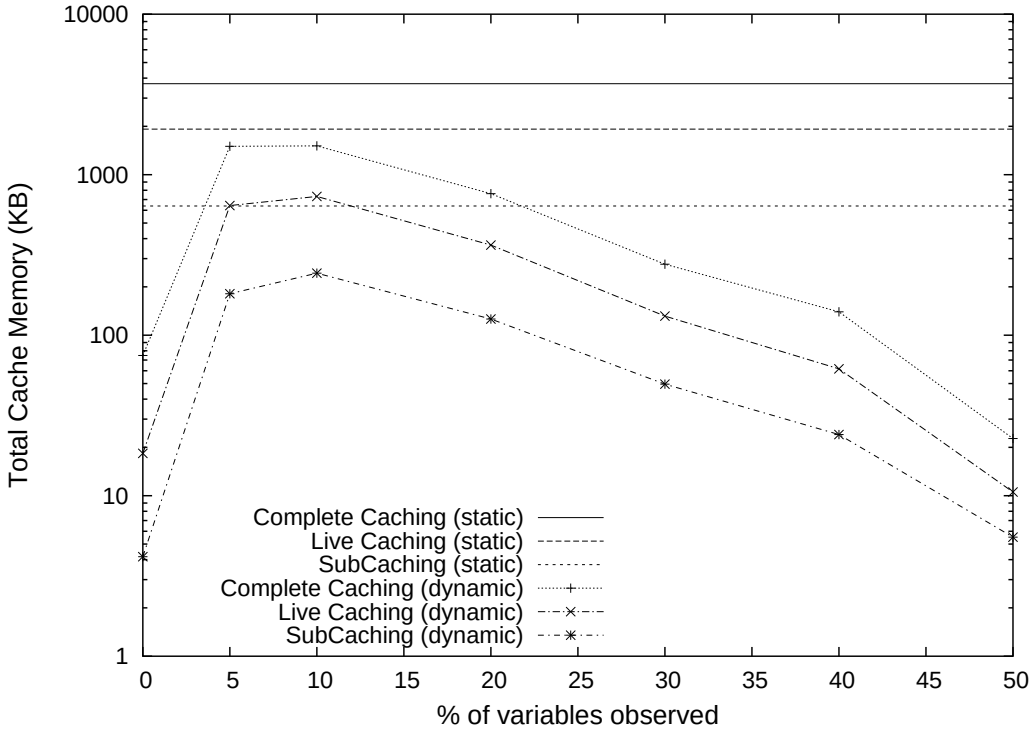


(a) Munin1

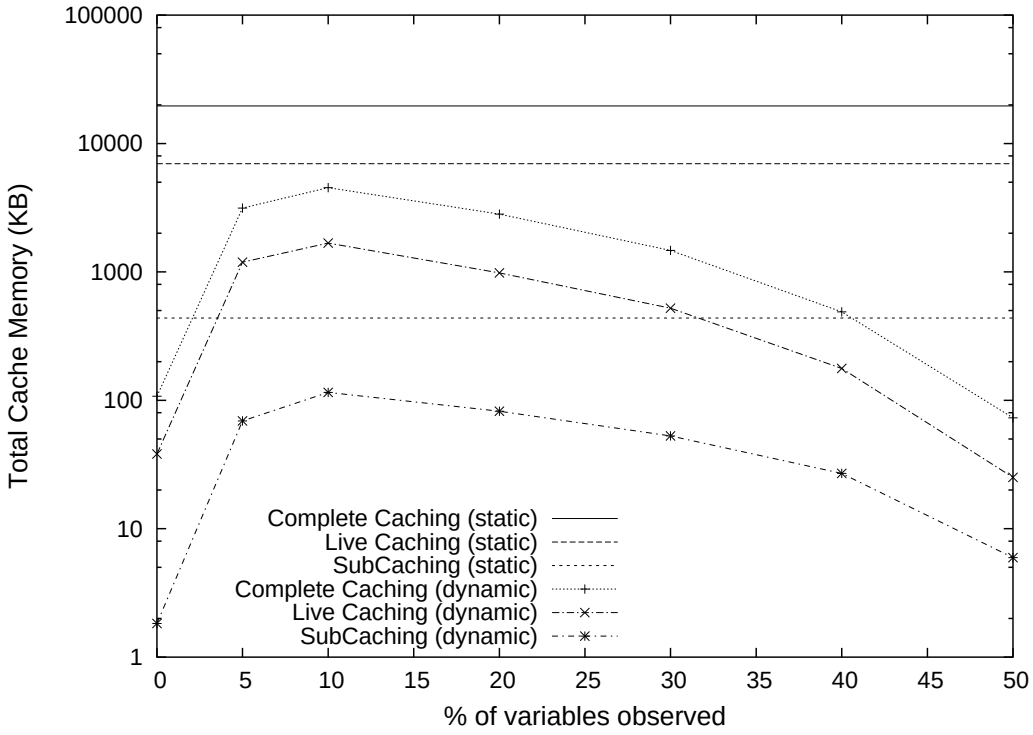


(b) Munin2

Figure 6.22: Memory requirements for caching in effective conditioning graphs vs. actual conditioning graphs (continued).

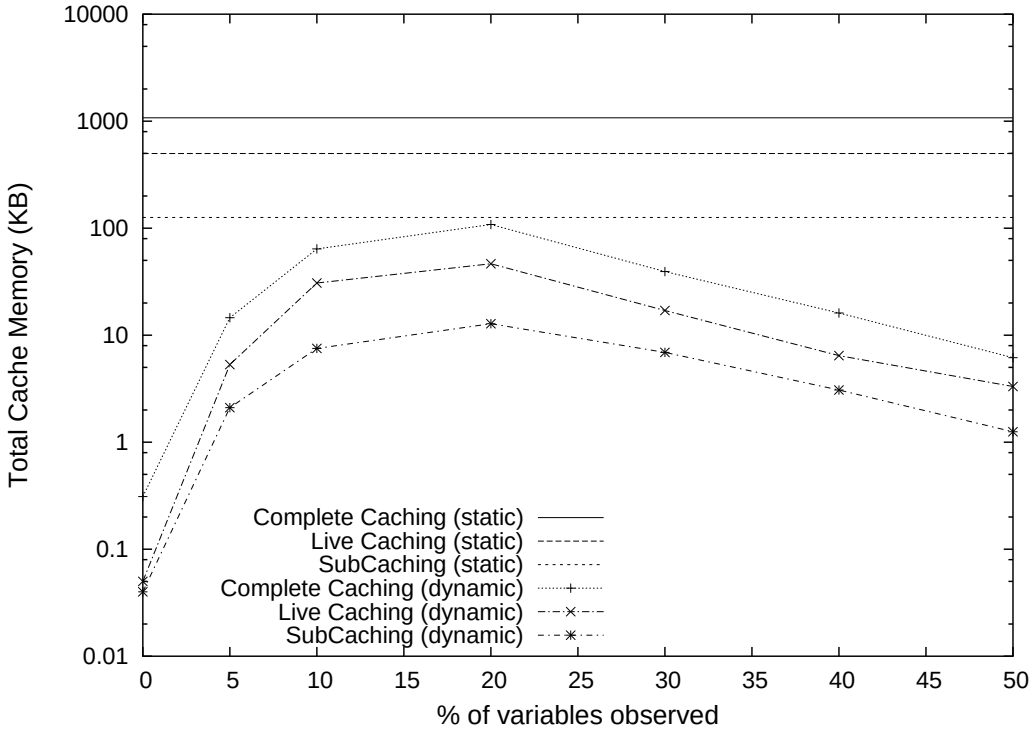


(a) Munin3

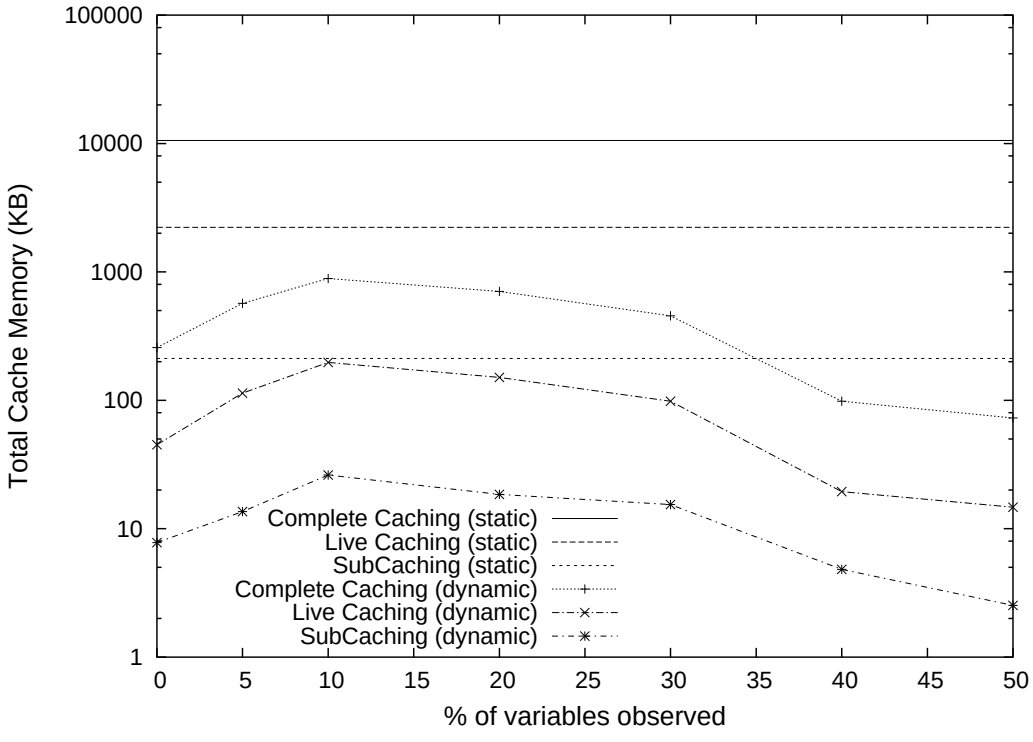


(b) Munin4

Figure 6.23: Memory requirements for caching in effective conditioning graphs vs. actual conditioning graphs (continued).



(a) Pigs



(b) Water

Figure 6.24: Memory requirements for caching in effective conditioning graphs vs. actual conditioning graphs (continued).

time can be quite modest. Seven of the ten tested networks required less than 256K of memory for subcaching on average; three of these networks (Mildew, Pigs, Water) required less than 50K, a remarkable result when compared to the memory requirements reported for JTP and VE. Recall that computing over the Link network and the Munin1 network using VE required 825 MB and 5770 MB, respectively; in the effective conditioning graph, Link required less than 3 MB for subcaching, while Munin1 required less than 10 MB.

6.3 Summary

This section examined caching as a means of optimizing runtimes when computing over conditioning graphs. Caching intermediate values avoids recomputation, allowing the conditioning graphs to compute probabilities in $\mathbf{O}(n \exp(w))$ time, which makes them asymptotically equivalent to JTP and VE.

We first considered caching when computing over the entire conditioning graph (that is, without excluding irrelevant variables). The techniques of caching in dtrees were adapted to accommodate conditioning graphs, with an emphasis on maintaining the goals of conditioning graphs in these algorithms (small, lightweight). We also introduced a new way to cache, subcaching, which reduces the space requirements of caching while not sacrificing the running time of the algorithm. An empirical comparison of the space requirements of conditioning graphs with caching showed a substantial reduction in comparison to JTP and VE, and the subcaching method showed even further improvement over the traditional methods.

We then considered caching in the effective conditioning graph. Irrelevant and observed variables were ignored in caches, which led to a further decrease in space requirements. Rather than allocating caches at compile-time, the space for caching was developed "on demand", in response to a query and evidence. The empirical results showed a further improvement in the amount of space required. The amount of memory required to compute over the networks used for testing was often less than 256K, on average.

Chapter 7

Conclusions

A conditioning graph is a recursive decomposition of a Bayesian network, which allows probability computation in $\mathbf{O}(n \exp(f))$ space, where f is the size of the largest family in the network. The inference algorithm is very small and simple, making it portable to almost any machine. Conditioning graphs abstract away the details of inference in Bayesian networks, and present themselves to the user as a simple data structure and a small algorithm, allowing the user accessibility to the code while not requiring any inference-specific expertise. This accessibility makes it possible for the time and space requirements to be assessed very quickly and precisely using tangible terms (bytes, instructions), rather than at a high-level (asymptotic notation). The user's interaction with the conditioning graph can be done completely through two simple function calls, *SetEvidence* and *Query*.

The time complexity of inference using conditioning graphs is exponential in the height of their underlying elimination tree (when no caching is employed). Hence, reducing its height is an important compile-time step. We showed how to derive conditioning graphs from dtrees in linear time. This conversion allowed us to take advantage of pre-existing methods for building shallow dtrees.

We proposed a new class of heuristics for constructing elimination trees directly from a variable ordering. These heuristics were based on the popular *min-size* and *min-fill* heuristics for generating good variable orderings for triangulation. The new heuristics incorporate a current cost component, which allows the heuristic to project and attempt to minimize the eventual height of the elimination tree. Using these heuristics proved to be superior to the dtree methods. These heuristics can also be applied to building shallow dtrees (through a reversal of the conversion process from

dtrees to elimination trees), and are most appropriate where no caching is performed.

We demonstrated how to optimize the conditioning graphs structure offline, using application-specific information, at the cost of constant memory per variable. Specifically, if the existence of sensor variables (always observable) is known, then these variables can be maintained separately from the primary structure (elimination tree), creating an exponential speedup in some cases (when the height of the tree is reduced). If we know a set of query variables in advance, then we can perform partial elimination during construction, further reducing the amount of computation necessary at runtime. Also, certain portions of the conditioning graph can be removed completely if they do not contain any query or evidence variables. The major advantage of these compile-time optimizations is that they are computed offline, and therefore their amortized computation costs are negligible.

We further extended the model to exploit this application-specific information at runtime. One of these optimizations, *hoods*, allowed evidence variables to be dynamically removed/replaced into the primary structure in linear time. We also defined the *effective conditioning graph* (the conditioning graph with barren and d-separated variables ignored) and presented algorithms for determining the effective conditioning graph and computing over it. These optimizations substantially reduced the runtime of inference over conditioning graphs, making it more competitive with VE, while adding space that is linear in the number of nodes and arcs in the Bayesian network.

Finally, we demonstrated how caching methods for recursive decompositions can be applied to conditioning graphs. Caching eliminates repeated computation, and allows calculation of probabilities in $\mathbf{O}(n \exp(w))$ time, equivalent to JTP and VE. We demonstrated *subcaching*, a new method for reducing the size of the caches at each node. We also demonstrated methods for eliminating irrelevant variables from the cache-domains of each node, which further reduced the space required for caching, without affecting the runtime of computation. Both subcaching and pruning irrelevant information showed a substantial improvement over previous methods for dtrees, in terms of space requirements.

7.1 Future Work

Throughout this disseration, our function library has been designed to calculate probabilities over evidence contexts, for eventual calculation of posterior probabilities. While this problem is the most common one of inference in Bayesian networks, Bayesian networks are also used for other tasks. Two of these tasks are called *Most Probable Explanation* (MPE) and *Maximal A Posteriori* (MAP) computations. MPE finds the context of all unobserved variables in the network consistent with the current context of the evidence that has the highest probability. MAP also finds the most likely context, but only for a subset of the unobserved variables (MPE is a special case of MAP). MPE can be solved using a modified version of VE in a two-stage process: (1) calculate the probability of the most likely state, and (2) generate the context of that most likely state from that probability calculation. Calculating the probability of the most likely state is a trivial extension to the *Query* algorithm: replace the plus operator (Figure 3.8, Line 15) with the *max* operator. However, the second phase of the algorithm assumes that the intermediate distributions were stored. Where no caching is used, none of the intermediate values exist, and would subsequently have to be recalculated. This means the conditioning graph algorithm would have to be run n times, once for each node, which may make it infeasible for all but the smallest models. MAP further complicates the algorithm by restricting the ordering in which variables can be eliminated. This would have implications on the conditioning graph structure, and subsequently its complexity. We plan to examine these two problems, and attempt to find a method for the calculation of these values, even when no caching is used.

The methods of this dissertation assume a Bayesian network with finite discrete variable domains. While this is a common application, it is certainly not the only graphical model for uncertain reasoning. Many applications, especially those involving physical sensor values, have variables whose domains are continuous or very large (e.g., velocity). Discretizing the variable domains is one option, but *continuous Bayesian network* models [48, 58] allow for continuous parameterized distributions.

Working with continuous variables avoids the errors introduced by discretization, and computing probabilities from continuous models is often more efficient than in discrete models. Hence, a conditioning graph model that works with continuous variables is desirable, and is the topic of future research.

We would also like to apply conditioning graphs to models outside of standard Bayesian networks. One of the primary goals of probabilistic reasoning is for decision making. While we can use the probabilities computed from the model to make decisions in some secondary model, *Influence Diagrams* [33, 59] incorporate decision and value nodes directly into the Bayesian network. Calculations over an influence diagram are similar to those in a Bayesian network, but with the goal of directly computing optimal policies (decisions that maximize the expected utility). Incorporating the decision-making process directly into the model is desirable, and hence compiling influence diagrams into conditioning graphs is another of our future goals.

Further research is required for the most efficient compilation of conditioning graphs that employ caching. When caching, the time complexity of a conditioning graph becomes a function of the width of the variable ordering used to construct it. In this case, a more appropriate strategy is to minimize the width of the variable ordering, rather than the height of the elimination tree. However, for mixed models, where not all values are cached, a mix of both would possibly be advantageous, where the complexity is not necessarily a function of height alone. Further research into heuristics for variable ordering is needed to determine such a strategy.

There is room for improvement in the heuristics of Chapter 4. Recall that for our experiments, when choosing which variable to eliminate next, ties in the estimation of the height of an elimination tree were broken arbitrarily, and the average of 50 runs was reported. In some cases, the standard deviation of these runs was high: different choices resulted in a difference of height that in some cases exceeded 2 or 3 variables. This effect was especially noticeable when choosing the earlier variables in the ordering. Because a linear reduction in the height results in an exponential increase in efficiency, we would like to discover a heuristic that finds the better cases when faced with a tie, ultimately improving the new heuristic approach.

The techniques of this paper reduce overhead due to large software libraries and runtime space. However, the size of the actual Bayesian network can be a limiting factor, as the number of CPT values is exponential on the network size. We examined compile-time pruning methods in Chapter 5, however, this assumes advance knowledge on query and evidence nodes, and these pruning techniques are not guaranteed to sufficiently reduce model size to fit the environment. The issue of large CPTs is of great concern in practical application. Disjunctive interaction models [50] (NOISY-OR) reduce the space requirements of CPTs from exponential to linear. Models that exploit context-specific independence [9, 54] and causal independence [66] can dramatically reduce the space requirements of general CPTs by exploiting local structure. Incorporating these methods into our model will require both structural change (to the graph itself) and functional change (to the algorithms), and represents another important avenue of research.

While the memory requirements of the conditioning graph are quite minimal, the optimizations introduced in the last three chapters extend these requirements. Most of the optimizations provided considerable speedup while requiring linear space on the number of nodes and arcs in the Bayesian network. Caching requires exponential space, but can produce runtimes equivalent to JTP and VE. When we do not have enough space to incorporate all of the optimizations, we have to decide what the best use of our space is. That is, do we use the available space to cache, or to store indexing parameters, or to mark irrelevant information. This is a second-order optimization problem: choosing which optimizations to apply, in order to maximize the expected benefit. We expect such an optimization problem to be hard, so we would like to discover techniques in order to produce good solutions.

We showed how subcaching provides a substantial reduction in the amount of memory used. However, it may not be enough to make the model small enough in some applications. In these cases, a partial caching scheme might be more appropriate. As mentioned, determining the best subset of nodes to cache at is a search problem. Subcaching complicates the search slightly, as the decision to cache at a particular node affects the size of the cache at lower nodes. Hence, modifying these

search algorithms to accommodate subcaching is also a topic of future research.

We would like to extend conditioning graphs to exploit determinism in Bayesian networks. Deterministic variables in a Bayesian network are characterized by zeros and ones in their CPT. Certain inference algorithms for Bayesian networks have been modified to exploit determinism for faster runtimes [24, 39]. We are currently examining modifications to the algorithms for computing over conditioning graphs, in order to exploit determinism and improve the efficiency of our approach.

Appendix A

Proof of Theorem 3.1.1

The proof of Theorem 3.1.1 requires notation for probability distributions and algorithmic function calls. Up to this point, we have separated variable instantations in a context with a comma. To avoid confusion from notation, we will use a comma to separate parameters in a algorithmic function call, and the $\wedge$ operator to separate variable instantations in a context.

Theorem 3.1.1. Given a Bayesian network $\langle \mathbf{X}, \mathbf{\Phi} \rangle$, an associated elimination tree T, and a variable $X_q \in \mathbf{V}$:

$$P(X_q = x_q | \mathbf{C} = \mathbf{c}) = \alpha \mathcal{P}(T, \{x_q\} \wedge \mathbf{c}) \tag{A.1}$$

where $\alpha = P(\mathbf{c})^{-1}$ is a normalization constant.

Given an elimination treenode T, recall that if T is an internal node, then X_T represents the variable labeling T, and ch_T represents the child nodes of T. If T is a leaf node, then ϕ_T represents the CPT at T. We extend our notation as follows: let $\mathbf{\Phi}_T$ represent the set of all CPTs in the leaves at or below T. As well, let $\mathbf{X}_T$ represent the variables labeling the internal nodes at or below T. Finally, we will write $\mathbf{X}_T - \mathbf{C}$ to represent the set of variables in $\mathbf{X}_T$ excluding any from $\mathbf{C}$.

Lemma A.0.1. *Given an elimination tree T and a context $\mathbf{c}$:*

$$\mathcal{P}(T, \mathbf{c}) = \sum_{\mathbf{y} \in \mathcal{D}(\mathbf{Y})} \prod_{\phi \in \mathbf{\Phi}_T} \phi(\mathbf{y} \wedge \mathbf{c}) \tag{A.2}$$

where $\mathbf{Y} = \mathbf{X}_T - \mathbf{C}$.

Proof. (By induction on T) The base case occurs when T is a leaf node. In this case,

$\mathbf{X}_T$ is empty, and $\mathbf{\Phi}_T = \{\phi_T\}$. So the summation and product are trivial, resulting in $\mathcal{P}(T, \mathbf{c}) = \phi_T$. The algorithm returns ϕ_T when T is a leaf node (the first conditional block in the algorithm), so the case holds.

The inductive step has two cases:

1. T is an internal node, where $X_T \in \mathbf{C}$. The algorithm $\mathcal{P}$ computes:

$$\mathcal{P}(T, \mathbf{c}) = \prod_{T' \in ch_T} \mathcal{P}(T', \mathbf{c}) \tag{A.3}$$

Using the inductive hypothesis, we can rewrite $\mathcal{P}(T', \mathbf{c})$ as follows:

$$\mathcal{P}(T, \mathbf{c}) = \prod_{T' \in ch_T} \sum_{\mathbf{u}' \in \mathcal{D}(\mathbf{U}')} \prod_{\phi \in \mathbf{\Phi}_{T'}} \phi(\mathbf{u}' \wedge \mathbf{c}) \tag{A.4}$$

where $\mathbf{U}' = \mathbf{X}_{T'} - \mathbf{C}$. Because each $\mathbf{U}'$ is disjoint, we can rewrite the product of sums as the sum of products, by taking the union of all of the variables in the summations. Let $\mathbf{U} = \bigcup_{T' \in ch_T} \mathbf{U}'$:

$$\mathcal{P}(T, \mathbf{c}) = \sum_{\mathbf{u} \in \mathcal{D}(\mathbf{U})} \prod_{T' \in ch_T} \prod_{\phi \in \mathbf{\Phi}_{T'}} \phi(\mathbf{u} \wedge \mathbf{c}) \tag{A.5}$$

Since $\mathbf{X}_T$ is defined as the set of variables labeling the internal nodes of the tree below node T, and since $X_T \in \mathbf{C}$, it follows that $\mathbf{U} = \mathbf{X}_T - \mathbf{C}$.

Combining the two products gives:

$$\mathcal{P}(T, \mathbf{c}) = \sum_{\mathbf{u} \in \mathcal{D}(\mathbf{U})} \prod_{\phi \in \mathbf{\Phi}'} \phi(\mathbf{u} \wedge \mathbf{c}) \tag{A.6}$$

where $\mathbf{\Phi}' = \bigcup_{T' \in ch_T} \mathbf{\Phi}_{T'} = \mathbf{\Phi}_T$, which proves the case.

2. T is an internal node, where $X_T \notin \mathbf{C}$. Algorithm $\mathcal{P}$ computes:

$$\mathcal{P}(T, \mathbf{c}) = \sum_{x \in \mathcal{D}(X_T)} \mathcal{P}(T, \{x\} \wedge \mathbf{c}) \tag{A.7}$$

Using the inductive hypothesis, we can show:

$$\mathcal{P}(T, \{x\} \wedge \mathbf{c}) = \sum_{\mathbf{y}' \in \mathcal{D}(\mathbf{Y}')} \prod_{\phi \in \mathbf{\Phi}_T} \phi(\mathbf{y}' \wedge \{x\} \wedge \mathbf{c}) \tag{A.8}$$

where $\mathbf{Y}' = \mathbf{X}_T - (\{X_T\} \bigcup \mathbf{C})$, that is, the set of internal variables in tree T, excluding X_T and any variable in $\mathbf{C}$. Therefore:

$$\mathcal{P}(T, \mathbf{c}) = \sum_{x \in \mathcal{D}(X_T)} \sum_{\mathbf{y}' \in \mathcal{D}(\mathbf{Y}')} \prod_{\phi \in \mathbf{\Phi}_T} \phi(\mathbf{y}' \wedge \{x\} \wedge \mathbf{c}) \tag{A.9}$$

The two summations can be combined, since the second one does not sum over X_T.

$$\mathcal{P}(T, \mathbf{c}) = \sum_{\mathbf{y} \in \mathcal{D}(\mathbf{Y})} \prod_{\phi \in \mathbf{\Phi}_T} \phi(\mathbf{y}, \mathbf{c}) \tag{A.10}$$

□

From this lemma, we can construct our proof of the Theorem:

Proof. From the lemma, we know that:

$$\mathcal{P}(T, \{x_q\} \wedge \mathbf{c}) = \sum_{\mathbf{y} \in \mathcal{D}(\mathbf{Y})} \prod_{\phi \in \mathbf{\Phi}_T} \phi(\mathbf{y} \wedge \{x_q\} \wedge \mathbf{c}) \tag{A.11}$$

where $\mathbf{Y} = \mathbf{X}_T - (\{X_q\} \bigcup \mathbf{C})$.

Since T refers to the elimination tree associated with the network, $\mathbf{X}_T = \mathbf{X}$, and $\mathbf{\Phi}_T = \mathbf{\Phi}$.

$$\mathcal{P}(T, \{x_q\} \wedge \mathbf{c}) = \sum_{\mathbf{y} \in \mathcal{D}(\mathbf{Y})} \prod_{\phi \in \mathbf{\Phi}} \phi(\mathbf{y} \wedge \{x_q\} \wedge \mathbf{c}) \tag{A.12}$$

where $\mathbf{Y} = \mathbf{X} - (\{X_q\} \bigcup \mathbf{C})$. This equation expresses the summation over variables not in $\mathbf{C}$ or X_q of the factorization defined by the Bayesian network; in other words, $\mathcal{P}(T, \{x_q\} \wedge \mathbf{c}) = P(\{x_q\}, \mathbf{c})$. Dividing this result by $\alpha = P(\mathbf{c})$ gives our result.

□

Appendix B
A C Implementation of Conditioning Graphs

This appendix contains a C implementation of the conditioning graph methods illustrated in Chapter 3. The motivation behind this implementation was to demonstrate the translation of the algorithm to an actual programming language, show a possible implementation and interface, and compare it to the resource requirements of other methods. We chose the C programming language because of its low-level nature and simplicity.

B.1 Node Representation

To implement the conditioning graph *Node* in C, we use a *struct*. We require a variable for each member of the node. A pseudocode version of the struct looks like this:

```
struct NODE {

/*** Leaf Node Variables ***/
  cpt: Array of DOUBLE;
  pos: INTEGER;

/*** Int. Node Variables ***/
  size: INTEGER;
  value: INTEGER;
```

```
  primary: Array of NODE;
  secondary: Array of NODE;

};
```

To maximize space efficiency, some of the variables are nested inside unions and inner structs (since we do not use the variables for an internal node and leaf node simultaneously).

We assume that the cardinality of the variables to be between 1 and 254. Therefore, the *size* and *value* variables of *Node* are stored as unsigned chars. We also assume that the number of probabilites per CPT can be stored in an integer. The type of the array for storing the probabilities is typedef'd, so that this can be easily changed for added or reduced precision. The primary and secondary vectors at each node are null-terminated arrays of pointers.

```
/*  cg.h                                                    */
/*  (c) 2006 by Kevin Grant, University of Saskatchewan  */
/*  cg.h stores the implementation functions for         */
/*  conditioning graphs, as outlined in Chapter 3 of my  */
/*  thesis.                                                 */

typedef double prob;

struct cg_node {
  union {
    struct {
      unsigned char size;               /*  N.size      */
      unsigned char value;              /*  N.value     */
    };
    int pos;                            /*  N.pos       */
  };
```

```
  struct cg_node **primary;              /*  N.primary   */

  union {
    struct cg_node **secondary;          /*  N.secondary */
    prob *cpt;                           /*  N.cpt       */
  };

} ;
```

B.2 Inference Functions

The implementations of the functions *SetEvidence* and *Query* are straightforward. We define a variable whose value is not known to take on value `0xFF`. Note that the line numbers are given to show the line-by-line translation from the specification given in Figures 3.8 and 3.9.

```
/*  cg.h (continued)                                        */


#define UNKNOWN 255

typedef struct cg_node cgn;

void set_evidence(cgn *N, unsigned char value) {
  N->value = value;                                        /*  1 */
}

prob query(cgn *N) {
  cgn **P, **S;
  prob total;
```

```
  if (N->primary == 0)                                    /*  1 */
    return N->cpt[N->pos];                                /*  2 */

  else if (N->value != UNKNOWN) {                         /*  3 */

    for (S = N->secondary; *S; S++)                       /*  4 */
       (*S)->pos = (*S)->pos * N->size + N->value;        /*  5 */

    total = 1;                                            /*  6 */

    for (P = N->primary; (*P); P++)                       /*  7 */
      total = total * query(*P);                          /*  8 */

    for (S = N->secondary; *S; S++)                       /*  9 */
      (*S)->pos = (*S)->pos / N->size;                    /* 10 */

    return total;                                         /* 11 */

  } else {                                                /* 12 */

    total = 0;                                            /* 13 */

    for (N->value = 0; N->value < N->size; ++N->value)    /* 14 */
      total = total + query(N);                           /* 15 */

    N->value = UNKNOWN;                                   /* 16 */
    return total;                                         /* 17 */

  }
}
```

B.3 Compilation

Once the user has defined a node representation and implemented the inference functions, the Bayesian network must be compiled into this representation. The implementation of this section is just one of the many ways that this could be implemented.

The function *set_evidence* requires that we have access to each node (unlike *query*, which requires access only to the root node of the structure). We store a linear array of pointers to each variable, and assign an integer index to each variable. The variable name of the array is *bn_<network name>*, where *<network name>* is the name of the network. The array allows subscript access to the internal nodes through the index. So from our example, if we observed the variable $ALARM$ to have value FALSE (0), we would call the following:

```
set_evidence(&bn_fire[ALARM], 0);  /* ALARM = false */
```

Once the context has been set, we can query its probability through the statement:

```
prob pr = query(bn_fire);  /* pr = P(e) */
```

The code is as follows:

```
#include "cg.h"

#define BN_FIRE_SIZE 6

cgn bn_fire[BN_FIRE_SIZE];              /*  internal node storage  */
cgn bn_fire_leaf[BN_FIRE_SIZE];         /*  leaf node storage      */

enum {ALARM=0, FIRE, LEAVING, SMOKE, TAMPERING, REPORT};

void initialize_bn_fire() {
```

```
 int i;

/**** Internal Nodes ****/

  bn_fire[ALARM].size = 2;
  bn_fire[ALARM].primary = (cgn **)malloc(3 * sizeof(cgn *));
  bn_fire[ALARM].primary[0] = &bn_fire[FIRE];
  bn_fire[ALARM].primary[1] = &bn_fire[LEAVING];
  bn_fire[ALARM].primary[2] = 0;

  bn_fire[ALARM].secondary = (cgn **)malloc(3 * sizeof(cgn *));
  bn_fire[ALARM].secondary[0] = &bn_fire_leaf[ALARM];
  bn_fire[ALARM].secondary[1] = &bn_fire_leaf[LEAVING];
  bn_fire[ALARM].secondary[2] = 0;

  bn_fire[FIRE].size = 2;
  bn_fire[FIRE].primary = (cgn **)malloc(4 * sizeof(cgn *));
  bn_fire[FIRE].primary[0] = &bn_fire[SMOKE];
  bn_fire[FIRE].primary[1] = &bn_fire[TAMPERING];
  bn_fire[FIRE].primary[2] = &bn_fire_leaf[FIRE];
  bn_fire[FIRE].primary[3] = 0;

  bn_fire[FIRE].secondary = (cgn **)malloc(4 * sizeof(cgn *));
  bn_fire[FIRE].secondary[0] = &bn_fire_leaf[SMOKE];
  bn_fire[FIRE].secondary[1] = &bn_fire_leaf[ALARM];
  bn_fire[FIRE].secondary[2] = &bn_fire_leaf[FIRE];
  bn_fire[FIRE].secondary[3] = 0;

  bn_fire[LEAVING].size = 2;
  bn_fire[LEAVING].primary = (cgn **)malloc(3 * sizeof(cgn *));
  bn_fire[LEAVING].primary[0] = &bn_fire_leaf[LEAVING];
  bn_fire[LEAVING].primary[1] = &bn_fire[REPORT];
```

```
bn_fire[LEAVING].primary[2] = 0;

bn_fire[LEAVING].secondary = (cgn **)malloc(3 * sizeof(cgn *));
bn_fire[LEAVING].secondary[0] = &bn_fire_leaf[LEAVING];
bn_fire[LEAVING].secondary[1] = &bn_fire_leaf[REPORT];
bn_fire[LEAVING].secondary[2] = 0;

bn_fire[SMOKE].size = 2;
bn_fire[SMOKE].primary = (cgn **)malloc(2 * sizeof(cgn *));
bn_fire[SMOKE].primary[0] = &bn_fire_leaf[SMOKE];
bn_fire[SMOKE].primary[1] = 0;

bn_fire[SMOKE].secondary = (cgn **)malloc(2 * sizeof(cgn *));
bn_fire[SMOKE].secondary[0] = &bn_fire_leaf[SMOKE];
bn_fire[SMOKE].secondary[1] = 0;

bn_fire[TAMPERING].size = 2;
bn_fire[TAMPERING].primary = (cgn **)malloc(3 * sizeof(cgn *));
bn_fire[TAMPERING].primary[0] = &bn_fire_leaf[TAMPERING];
bn_fire[TAMPERING].primary[1] = &bn_fire_leaf[ALARM];
bn_fire[TAMPERING].primary[2] = 0;

bn_fire[TAMPERING].secondary = (cgn **)malloc(3 * sizeof(cgn *));
bn_fire[TAMPERING].secondary[0] = &bn_fire_leaf[TAMPERING];
bn_fire[TAMPERING].secondary[1] = &bn_fire_leaf[ALARM];
bn_fire[TAMPERING].secondary[2] = 0;

bn_fire[REPORT].size = 2;
bn_fire[REPORT].primary = (cgn **)malloc(2 * sizeof(cgn *));
bn_fire[REPORT].primary[0] = &bn_fire_leaf[REPORT];
bn_fire[REPORT].primary[1] = 0;
```

```
  bn_fire[REPORT].secondary = (cgn **)malloc(2 * sizeof(cgn *));
  bn_fire[REPORT].secondary[0] = &bn_fire_leaf[REPORT];
  bn_fire[REPORT].secondary[1] = 0;

  for (i = 0; i < BN_FIRE_SIZE; i++)
    bn_fire[i].value = UNKNOWN;      /* initialize var as unobserved */

/**** Leaf Nodes ****/

  bn_fire_leaf[REPORT].cpt = (prob *)malloc(4 * sizeof(prob));
  bn_fire_leaf[REPORT].cpt[0] = 0.99;
  bn_fire_leaf[REPORT].cpt[1] = 0.01;
  bn_fire_leaf[REPORT].cpt[2] = 0.25;
  bn_fire_leaf[REPORT].cpt[3] = 0.75;

  bn_fire_leaf[LEAVING].cpt = (prob *)malloc(4 * sizeof(prob));
  bn_fire_leaf[LEAVING].cpt[0] = 0.999;
  bn_fire_leaf[LEAVING].cpt[1] = 0.001;
  bn_fire_leaf[LEAVING].cpt[2] = 0.12;
  bn_fire_leaf[LEAVING].cpt[3] = 0.88;

  bn_fire_leaf[FIRE].cpt = (prob *)malloc(2 * sizeof(prob));
  bn_fire_leaf[FIRE].cpt[0] = 0.99;
  bn_fire_leaf[FIRE].cpt[1] = 0.01;

  bn_fire_leaf[ALARM].cpt = (prob *)malloc(8 * sizeof(prob));
  bn_fire_leaf[ALARM].cpt[0] = 0.9999;
  bn_fire_leaf[ALARM].cpt[1] = 0.15;
  bn_fire_leaf[ALARM].cpt[2] = 0.01;
  bn_fire_leaf[ALARM].cpt[3] = 0.5;
  bn_fire_leaf[ALARM].cpt[4] = 0.0001;
  bn_fire_leaf[ALARM].cpt[5] = 0.85;
```

```
  bn_fire_leaf[ALARM].cpt[6] = 0.99;
  bn_fire_leaf[ALARM].cpt[7] = 0.5;

  bn_fire_leaf[TAMPERING].cpt = (prob *)malloc(2 * sizeof(prob));
  bn_fire_leaf[TAMPERING].cpt[0] = 0.98;
  bn_fire_leaf[TAMPERING].cpt[1] = 0.02;

  bn_fire_leaf[SMOKE].cpt = (prob *)malloc(4 * sizeof(prob));
  bn_fire_leaf[SMOKE].cpt[0] = 0.99;
  bn_fire_leaf[SMOKE].cpt[1] = 0.01;
  bn_fire_leaf[SMOKE].cpt[2] = 0.1;
  bn_fire_leaf[SMOKE].cpt[3] = 0.9;

  for ( i = 0; i < BN_FIRE_SIZE; i++) {
   bn_fire_leaf[i].primary = 0;            /* designate node as leaf */
   bn_fire_leaf[i].pos     = 0;            /* initialize cpt index to 0 */
  }
}
```

B.4 Example

To demonstrate the use of this compilation, we give a small example. Suppose that given the *Fire* network, we wish to build a small monitoring system. This system will monitor the states of three variables: *Smoke*, *Alarm*, and *Leaving* and calculate the probabilities of a *Fire* and the probability that the Alarm has been tampered with (*Tampering*). If the probability of *Fire* is sufficiently high ($> 10\%$), it will alert the local Fire department. Likewise, if the probability of *Tampering* is sufficiently high ($> 25\%$), it will alert the local repair department.

To create this monitoring system, we write a function called *check*. *check* takes the current observed values of the three mentioned variables. If it deems the probability of *Fire* or *Tampering* to be sufficiently high, it alerts the appropriate response

unit. Active monitoring requires periodically observing the variables, and calling *check* with their values.

The C implementation of *check* is as follows:

```
void check(unsigned char alarm_val,
           unsigned char smoke_val,
           unsigned char leave_val) {

  printf("----------------\n");
  printf("System status:  Alarm = %d, Smoke = %d, Leaving = %d\n",
                                 alarm_val, smoke_val, leave_val);

  /* Apply the observations to the Bayesian network. */

  set_evidence(&bn_fire[ALARM], alarm_val);
  set_evidence(&bn_fire[SMOKE], smoke_val);
  set_evidence(&bn_fire[LEAVING], leave_val);

  /* Calculate the probability of the evidence */

  prob e = query(bn_fire);

  /* Set Fire = true */

  set_evidence(&bn_fire[FIRE], 1);

  /* Calculate the joint prob. of fire and the evidence */

  prob fe = query(bn_fire);

  printf("Probability of fire = %f", (fe / e));
```

```
  /* If the conditioninal prob. of fire > 10%, report it. */

  if ((fe / e) > 0.10)
      printf(", alerting local fire department.");

  /* Reset the value of fire. */

  set_evidence(&bn_fire[FIRE], UNKNOWN);

  /* Set Tampering = true */

  set_evidence(&bn_fire[TAMPERING], 1);

  /* Calculate the joint prob. of tampering and the evidence */

  prob te = query(bn_fire);

  printf("\nProbability of tampered system = %f", (te / e));

  /* If the conditioninal prob. of tampering > 25%, report it. */

  if ((te / e) > 0.25)
      printf(", alerting local repair shop.");

  /* Reset the value of Tampering */

  set_evidence(&bn_fire[TAMPERING], UNKNOWN);

  printf("\n----------------\n");

}
```

Appendix C
A MIPS Implementation of Conditioning Graphs

This appendix outlines a MIPS implementation of the conditioning graph methods illustrated in Chapter 3. The motivation behind this implementation was to empirically assess the memory footprint/time requirements of conditioning graph inference at a very low, very precise level. We chose MIPS as it is a common machine language amongst many computer scientists, it has a floating point unit, and it is a RISC processor (so it makes no assumptions on complex instructions).

C.1 Node Representation

As in the last chapter, a representation for the node must be chosen. However, unlike the last chapter, no explicit structure must be stored, just the implicit storage rules.

Leaf nodes are stored as a word representing the CPT index of the node (*N.pos*), followed by a list of the CPT entries (as a sequence of double values). For example, we represent the *Alarm* leaf node as follows:

```
alarm_leaf:     .word   0
                .double 0.9999, 0.15, 0.01, 0.5
                .double 0.0001, 0.85, 0.99, 0.5
```

Internal nodes are stored as a sequence of four bytes, representing variable size, variable value, number of secondary children, and number of primary children, respectively. These four bytes are followed by two lists: the addresses of the secondary

children, and the addresses of the primary children (each stored as a sequence of words). For example, the *Alarm* internal node is represented like this:

```
alarm:          .byte 2, 0xFF, 2, 2
                .word alarm_leaf, leaving_leaf
                .word fire, leaving
```

The entire compilation of the *Fire* network is as follows:

```
alarm:          .byte 2, 0xFF, 2, 2
                .word alarm_leaf, leaving_leaf
                .word fire, leaving

fire:           .byte 2, 0xFF, 3, 3
                .word smoke_leaf, alarm_leaf, fire_leaf
                .word smoke, tampering, fire_leaf

leaving:        .byte 2, 0xFF, 2, 2
                .word leaving_leaf, report_leaf
                .word leaving_leaf, report

smoke:          .byte 2, 0xFF, 1, 1
                .word smoke_leaf
                .word smoke_leaf

tampering:      .byte 2, 0xFF, 2, 2
                .word tampering_leaf, alarm_leaf
                .word tampering_leaf, alarm_leaf

report:         .byte 2, 0xFF, 1, 1
                .word report_leaf
                .word report_leaf
```

```
alarm_leaf:     .word 0
                .double 0.9999, 0.15, 0.01, 0.5
                .double 0.0001, 0.85, 0.99, 0.5

fire_leaf:      .word 0
                .double 0.99, 0.01

leaving_leaf:   .word 0
                .double 0.999, 0.001, 0.12, 0.88

smoke_leaf:     .word 0
                .double 0.99, 0.01, 0.1, 0.9

tampering_leaf: .word 0
                .double 0.98, 0.02

report_leaf:    .word 0
                .double 0.99, 0.01, 0.25, 0.75
```

C.2 Inference Functions

The implementations of the functions *Query* and *SetEvidence* are given in this section. While not as direct a translation as the C implementation of the last chapter, these functions can still be easily recognized as implementations of the algorithms in Figures 3.8 and 3.9. We adopt the following conventions:

- For loops are implemented using branches and jumps.
- Arithmetic calculations must be performed with registers. Hence, all values must first be loaded out of memory into registers.
- Variable values must be stored on the stack, to ensure consistency during re-

cursion.

We make one major change in our implementation. Leaf nodes are stored in higher memory addresses than internal nodes, which means that we can check for a leaf node (Line 01 of the *Query* algorithm) by checking whether the address of the node is sufficiently high. Hence, we our *Query* algorithm takes two addresses: the address of the current node being processed, and the address of the leaf node with the smallest address. To check whether the current node being processed is a leaf node, we need only compare whether its address is greater than or equal to the second parameter.

To summarize, when a variable is observed, we set its value by loading the node's address and the value into parameters $a0 and $a1, and calling *set_evidence*. For example, to set *Alarm* to true (value 1), we would issue the following commands:

```
la      $a0, alarm          # param. 1 = alarm node
li      $a1, 1              # param. 2 = 1
jal     set_evidence        # call set_evidence
```

To query the probability of the current context, we load the root node and the lowest-addressed leaf node into parameters $a0 and $a1, and call query:

```
la      $a0, alarm          # param. 1 = root node (alarm)
la      $a1, alarm_leaf     # param. 2 = alarm leaf node
jal     query               # call query
```

The entire listing for the functions is as follows:

```
#set_evidence(N, i)
#Input:    $a0 - address of node to set context to
#          $a1 - new value of node (0xFF if unobserved)

set_evidence:
  sb $a1, 1($a0)
```

```
jr $ra

# Query(N)
# Input:   $a0 - address of current node (N)
#          $a1 - address of beginning leaf node
# Output:  $f0 - probability of current context

query:
       subu  $sp,$sp,36
       sw    $t0, 0($sp)        # recursive function requires
       sw    $t1, 4($sp)        # that we store variables on
       sw    $t2, 8($sp)        # stack
       sw    $t3, 12($sp)       #
       sw    $t4, 16($sp)       #
       s.d   $f2, 20($sp)       #
       sw    $ra, 28($sp)       #
       sw    $fp, 32($sp)       #

       bgt   $a1, $a0, $Q1      # if N is a leaf node
       lw    $t0, 0($a0)        #    $t0 = N.pos
       addi  $t1, $a0, 8        #    $t1 = N.cpt
       sll   $t0, $t0, 3        #
       add   $t1 $t1 $t0        #    $t1 = N.cpt + N.pos
       l.d   $f2, 0($t1)        #    $f2 = N.cpt[N.pos]
       jr    $Q0                #    goto pop stack and exit

$Q1:   move  $t0, $a0           # $t0 = N
       lb    $t1, 1($t0)        # $t1 = N.value
       lb    $t2, 0($t0)        # $t2 = N.size
       bltz  $t1, $Q2           # if N.value <> UNKNOWN

       lb    $t3, 2($t0)        #   $t3 = # of secondary pointers
```

```
        addi  $t4, $t0, 4        #   $t4 = N.secondary

$L0:    blez  $t3, $L1           #   for each S in N.secondary
        lw    $t5, 0($t4)        #     $t5 = S
        lw    $t6, 0($t5)        #     $t6 = S.pos
        mul   $t6, $t6, $t2      #     S.pos = S.pos * N.size
        add   $t6, $t6, $t1      #     S.pos = S.pos + N.value
        sw    $t6, 0($t5)        #     store new value of S.pos
        addi  $t3, $t3, -1       #
        addi  $t4, $t4, 4        #
        jr    $L0                #     next S

$L1:    l.d   $f2, c_one         #   Total = 1
        lb    $t3, 3($t0)        #   $t3 = number of primary pointers

$L2:    blez  $t3, $P1           #   for each P in N.primary
        lw    $a0, 0($t4)        #     $a0 = P
        jal   query              #     $f2 = query(P)
        mul.d $f2, $f2, $f0      #     Total = Total * Query(P)
        addi  $t3, $t3, -1       #
        addi  $t4, $t4, 4        #
        jr    $L2                #     Next P

$P1:    lb    $t3, 2($t0)        #  $t3 = number of secondary pointers
        addi  $t4, $t0, 4        #  $t4 = N.secondary

$L3:    blez  $t3, $Q0           #   for each S in N.secondary
        lw    $t5, 0($t4)        #     $t5 = S
        lw    $t6, 0($t5)        #     $t6 = S.pos
        div   $t6, $t6, $t2      #     S.pos = S.pos / N.size
        sw    $t6, 0($t5)        #     store new value of S.pos
```

```
        addi  $t3, $t3, -1       #
        addi  $t4, $t4, 4        #
        jr    $L3                #     next S
                                 #   goto pop stack and exit

                                 # if N.value = UNKONWN
$Q2:    l.d   $f2, c_zero        #   Total = 0
$Q3:    blez  $t2, $Q4           #   for N.value = N.size - 1 to 0
        addi  $t2, $t2, -1       #
        sb    $t2, 1($t0)        #      store N.value
        move  $a0, $t0           #      $a0 = N
        jal   query              #      $f2 - Query(N)
        add.d $f2, $f2, $f0      #      Total = Total + Query(N)
        jr    $Q3                #      Next N.value

$Q4:    li    $t2, 0xFF          #   N.value = UNKNOWN
        sb    $t2, 1($t0)        #   goto pop stack and exit

$Q0:    mov.d  $f0, $f2          # return Total in $f0

        lw     $t0, 0($sp)       # Pop values from the stack
        lw     $t1, 4($sp)       # before returning
        lw     $t2, 8($sp)       #
        lw     $t3, 12($sp)      #
        lw     $t4, 16($sp)      #
        l.d    $f2, 20($sp)      #
        lw     $ra, 28($sp)      #
        lw     $fp, 32($sp)      #
        addiu  $sp, $sp, 36      # Pop stack
        jr     $ra               # Return to caller
```

C.3 Example

To illustrate the use of this code, we reproduce the monitoring example from the last appendix, using our MIPS code. Recall that our monitoring system monitors three variables, and calculates the probabilities of a *Fire*, and that the alarm has been tampered with (*Tampering*). If the probability of *Fire* (*Tampering*) is sufficiently high, then the local fire department (repair shop) is alerted.

The function *check* is passed the values of *Alarm*, *Smoke*, and *Leaving* through registers $a0, $a1, and $a2. As with the C implementation, the program simply outputs strings indicating the state of the program. Note that outputting formatted text in MIPS is not nearly as elegant as it is in C. Therefore, many of the instructions in the following function are dedicated to output.

```
check: move     $t0, $a0          # store the observed
       move     $t1, $a1          # values in $t0-$t2
       move     $t2, $a2
       move     $t3, $ra

       li       $v0, 4            # print out the current context
       la       $a0, str_line     # of the observed variables
       syscall
       la       $a0, str_stat1
       syscall
       li       $v0, 1
       move     $a0, $t0
       syscall
       li       $v0, 4
       la       $a0, str_stat2
       syscall
       li       $v0, 1
       move     $a0, $t1
```

```
syscall
li       $v0, 4
la       $a0, str_stat3
syscall
li       $v0, 1
move     $a0, $t2
syscall

la       $a0, alarm           # set the value of alarm
move     $a1, $t0
jal      set_evidence

la       $a0, smoke           # set the value of smoke
move     $a1, $t1
jal      set_evidence

la       $a0, leaving         # set the value of leaving
move     $a1, $t2
jal      set_evidence

la       $a0, alarm           # calculate the probability
la       $a1, alarm_leaf      # of the evidence
jal      query

mov.d    $f4, $f0             # store the result in $f4

la       $a0, fire            # set Fire = true
li       $a1, 1
jal      set_evidence

la       $a0, alarm           # calculate the joint prob.
la       $a1, alarm_leaf      # of fire and the evidence
```

```
jal     query

div.d   $f6, $f0, $f4       # calculate the cond. prob.
                            # of fire given evidence
                            # store result in $f6

la      $a0, fire           # reset value of Fire
li      $a1, 0xff
jal     set_evidence

la      $a0, tampering      # set Tampering = true
li      $a1, 1
jal     set_evidence

la      $a0, alarm          # calculate the joint prob.
la      $a1, alarm_leaf     # of Tampering and evidence
jal     query

div.d   $f8, $f0, $f4       # calculate the cond. prob.
                            # of Tampering given evidence
                            # store result in $f8

la      $a0, tampering      # reset value of Tampering
li      $a1, 0xff
jal     set_evidence

li      $v0, 4 # report p(Fire)
la      $a0, str_fire1
syscall

li      $v0, 3
mov.d   $f12 $f6
```

```
        syscall

        l.d     $f10 c_10           # if p(Fire) > 0.10
        c.lt.d  $f10 $f6            # report to fire dept.
        bc1f    $R0                 #

        li      $v0, 4
        la      $a0, str_fire2
        syscall

$R0:    li      $v0, 4 # report p(Tampering)
        la      $a0, str_tamp1
        syscall

        li      $v0, 3
        mov.d   $f12 $f8
        syscall

        l.d     $f10 c_25           # if P(Tampering) > 0.25
        c.lt.d  $f10 $f8            # report to local repair
        bc1f    $R1

        li      $v0, 4
        la      $a0, str_tamp2
        syscall

$R1:    li      $v0, 4
        la      $a0, str_line
        syscall

        move    $ra $t3
        jr      $ra
```

References

[1] Bayesian network repository. http://compbio.cs.huji.ac.il/Repository/.

[2] B. Abramson, J. Brown, W. Edwards, A. Murphy, and R. L. Winkler. Hailfinder: A bayesian system for forecasting severe weather. *International Journal of Forecasting*, 12(1):57–71, 1996.

[3] D. Allen and A. Darwiche. New advances in inference by recursive conditioning. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 2–10, 2003.

[4] D. Allen and A. Darwiche. Optimal time–space tradeoff in probabilistic inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 969–975, 2003.

[5] D. Allen, A. Darwiche, and J. D. Park. A greedy algorithm for time-space tradeoff in probabilistic inference. In *Proceedings of the Second European Workshop on Probabilistic Graphical Models*, pages 1–8, 2004.

[6] S. K. Andersen, K. G. Olesen, F. V. Jensen, and F. Jensen. Hugin: A shell for building bayesian belief universes for expert systems. In *Readings in Uncertain Reasoning*, pages 332–337. Morgan Kaufmann Publishers Inc., 1990.

[7] S. Andreassen, F. V. Jensen, S. K. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A. R. Sørensen, A. Rosenfalck, and F. Jensen. MUNIN — an expert EMG assistant. In J. E. Desmedt, editor, *Computer-Aided Electromyography and Expert Systems*, chapter 21. Elsevier Science Publishers, 1989.

[8] A. Becker and D. Geiger. Approximation algorithms for the loop cutset problem. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 60–68, 1994.

[9] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 115–123, 1996.

[10] H. Chan and A. Darwiche. When do numbers really matter? *Journal of Artificial Intelligence Research*, 17:265–287, 2002.

[11] G. F. Cooper. Bayesian belief-network inference using recursive decomposition. Technical Report KSL-90-05, Knowledge Systems Laboratory, Stanford, CA, 94305, USA, 1990.

[12] G. F. Cooper. The Computational Complexity of Probabilistic Inference using Bayesian Belief Networks. *Artificial Intelligence*, 42:393–405, 1990.

[13] R. Cowell and A. Dawid. Fast retraction of evidence in a probabilistic expert system. *Statistics and Computing*, 2:37–40, 1992.

[14] P. Dagum and M. Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence*, 60:141–153, 1993.

[15] A. Darwiche. Recursive Conditioning: Any-space conditioning algorithm with treewidth-bounded complexity. *Artificial Intelligence*, pages 5–41, 2000.

[16] A. Darwiche. A differential approach to inference in bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

[17] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143*, pages 180–191. Springer-Verlag, 2001.

[18] A. Darwiche and G. Provan. Query dags: A practical paradigm for implementing belief network inference. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 203–210, 1996.

[19] A. Darwiche and G. Provan. A standard approach for optimizing belief network inference using query dags. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 116–123, 1997.

[20] R. Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference Algorithms. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 211–219, 1996.

[21] R. Dechter. Topological parameters for time-space tradeoff. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 220–227, 1996.

[22] R. Dechter. Mini-buckets: A general scheme for generating approximations in automated reasoning. In *IJCAI*, pages 1297–1303, 1997.

[23] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[24] R. Dechter and D. Larkin. Hybrid processing of beliefs and constraints. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 112–119, 2001.

[25] R. Dechter and I. Rish. Mini Buckets: A general scheme for bounded inference. *Journal of the ACM*, 50:107–153, 2003.

[26] F. Diez. Local Conditioning in Bayesian Networks. *Artificial Intelligence*, 87:1–20, 1996.

[27] J. Finn. Cautious propagation in bayesian networks. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 323–328, 1995.

[28] J. Finn and J. Frank. Optimal junction trees. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 360–366, 1994.

[29] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[30] D. Geiger, T. Verma, and J. Pearl. Identifying independence in Bayesian networks. *Networks*, 20:507–534, 1990.

[31] K. Grant and M. Horsch. Methods for Constructing Balanced Elimination Trees and Other Recursive Decompositions. In *Proceedings of the the Nineteenth International Florida Artificial Intelligence Research Society Conference*, 2006.

[32] E. Horvitz, H. Suermondt, and G. F. Cooper. Bounded conditioning: Flexible inference for decisions under scarce resources. In *Proceedings of the Fifth Conference on Uncertainty in Artificial Intelligence*, pages 182–193, 1989.

[33] R. Howard and J. Matheson. The principles and applications of decision analysis. Technical report, Strategic Decisions Group, CA, 1981.

[34] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[35] F. Jensen, S. Lauritzen, and K. Oleson. Bayesian Updating in Causal Probabilistic Networks by Local Computations. *Computational Statistics Quarterly*, 4:269–282, 1990.

[36] F. V. Jensen. *Introduction to Bayesian Networks*. Springer-Verlag New York, Inc., 1996.

[37] F. V. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag New York, Inc., 2001.

[38] U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical Report R 90-09, Dept. of Mathematics and Computer Science, Strandvejan, DK 9000 Aalborg, Denmark, 1990.

[39] D. Larkin and R. Dechter. Bayesian inference in the presence of determinism. In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, 2003.

[40] S. Lauritzen and D. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50:157–224, 1988.

[41] U. Lerner, R. Parr, D. Koller, and G. Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 531–537, 2000.

[42] I. Masao. Simultaneous computation of functions, partial derivates, and estimates of rounding error. *Japan Journal of Applied Mathematics*, 1:223–252, 1984.

[43] G. L. Miller and J. Reif. Parallel tree contraction and its application. In *Proceedings of the Twenty-Sixth IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[44] S. Monti and G. F. Cooper. Bounded recursive decomposition: a search-based method for belief-network inference under limited resources. *International Journal of Approximate Reasoning*, 15(1):49–75, 1996.

[45] K. Murphy, Y. Weiss, and M. Jordan. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 467–475, 1999.

[46] R. E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, April 2003.

[47] Netica. Norsys: Software corporation. *http://www.norsys.com/netica.html*, 2006.

[48] K. G. Olesen. Causal probabilistic networks with both discrete and continuous variables. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):275–279, 1993.

[49] J. Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29:241–288, 1986.

[50] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[51] M. Peot and R. Shachter. Fusion and propagation with multiple observations. *Artificial Intelligence*, 48:299–318, 1991.

[52] D. Poole. Exploiting contextual independence and approximation in belief network inference. Technical report, Dept. of Computer Science, UBC, 1997.

[53] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence*. Oxford University Press, 1998.

[54] D. Poole and N. Zhang. Exploiting contextual independence in probabilistic inference. *Journal of Artificial Intelligence Research*, 18:263–313, 2003.

[55] F. Ramos, F. Cozman, and J. Ide. Embedded bayesian networks: Anyspace, anytime probabilistic inference. In *Proceedings of the AAAI/KDD/UAI Workshop in Real-time Decision Support and Diagnosis Systems*, 2002.

[56] D. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32:597–609, 1974.

[57] S. Ross, A. Stig, and S. Peter. Global conditioning for probabilistic inference in belief networks. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 514–522, 1994.

[58] S. Roweis and Z. Ghahramani. A unifying review of linear gaussian models. *Neural Comput.*, 11(2):305–345, 1999.

[59] R. D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871–882, 1986.

[60] R. D. Shachter. Probabilistic inference and influence diagrams. *Operations Research*, 36(4):589–604, 1988.

[61] R. D. Shachter. Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 480–487, 1998.

[62] J. Stillman. On heuristics for finding loop cutsets in multiply connected belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 233–243, 1991.

[63] H. Suermondt and G. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. *Computers and Biomedical Research*, 24:453–475, 1991.

[64] T. Verma and J. Pearl. Causal networks: Semantics and expressiveness. In *Proceedings of the 4th Annual Conference on Uncertainty in Artificial Intelligence (UAI-88)*, 1988.

[65] N. Zhang and D. Poole. A Simple Approach to Bayesian Network Computations. In *Proceedings of the Tenth Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.

[66] N. Zhang and D. Poole. Exploiting Causal Independence in Bayesian Network Inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.